Felix - a Simulation-Tool for Neural Networks

(and Dynamical Systems)

USER GUIDE

Thomas Wennekers

Centre for Theoretical and Computational Neuroscience
University of Plymouth
PL4 8AA Plymouth, Devon, United Kingdom

March 10, 2007

Dear valued Reader

This is the User Guide of "Felix", a simulation environment for neural networks and dynamical systems. It is C-based and provides a simple to use graphical interface as well as real time control of simulation parameters. The main aim of the tool is to simplify the implementation and simulation of distributed neural networks consisting of either homogeneous pools or 2-dimensional layers of simple spiking neurons. Other, more general dynamical systems can be implemented and visualised as well, and several examples are provided (coupled map lattice, coupled Roessler oscillators). The simulation of conductance-based neuron types is possible but only marginally supported.

The tool can make use of code-parallelisation on three levels: single CPU vectorisation using BLAS-SSE2, SMP-shared memory parallelism via OpenMP (threads), and the message passing interface (MPI) for computer clusters. Hybrid BLAS/OpenMP/MPI code is possible, e.g., for use on SMP-clusters. Felix can be downloaded from http://www.pion.ac.uk, which provides run-time libraries, the development tool, and a couple of examples. Source code is also available and, beside on Linux single- and multi-processor computers, and Linux Beowulf clusters, can be compiled and run on Windows using the Cygwin-Linux emulator.

Have fun
Thomas Wennekers

Contents

Chapter 1

Introduction

1.1 Overview

This is a preliminary version of a User Guide for “Felix” - A simulation tool for neural networks and dynamical systems. It is currently being written. This introduction, the quick-start guide in section 2, sections 3 about the graphical user interface and 5 about file I/O, the description of the main function libraries in section 4, and the appendix about installation A are something like in a readable state. The examples (section 7) and the section about parallel Felix extensions 6 are still mostly empty or bad. You would probably want to consult the examples that come with Felix directly, if you think about using the tool and want to learn more about how to do so. Serial and parallel example programs are available.

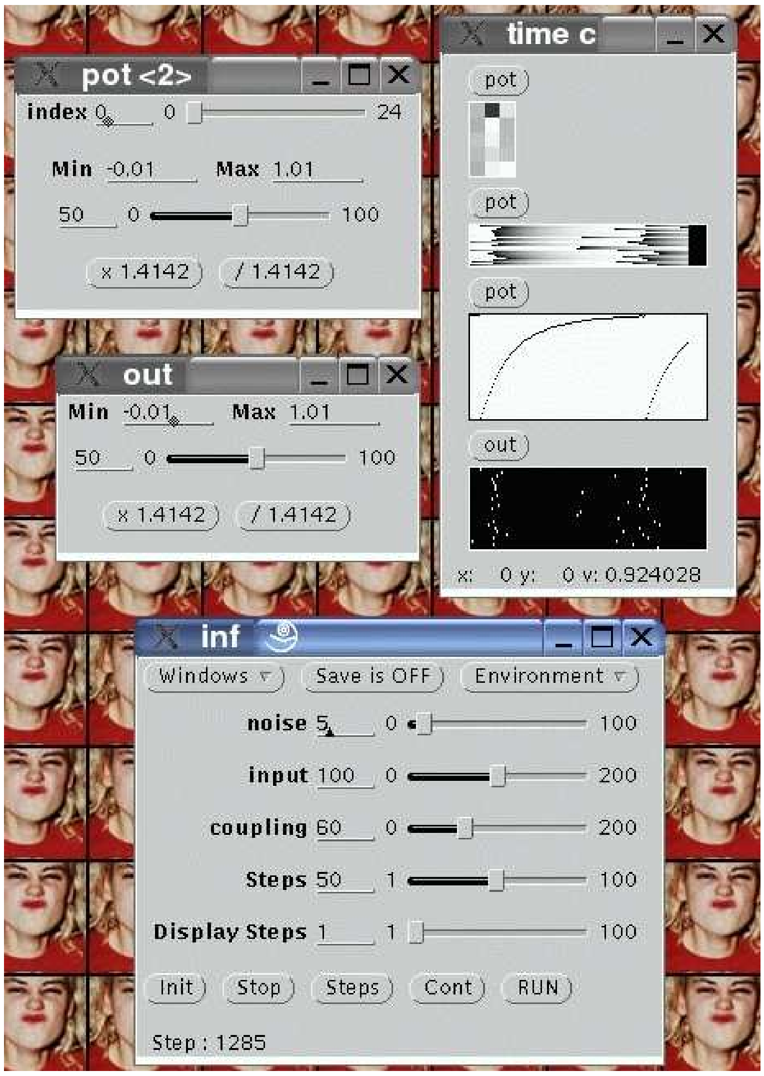
Felix is a development tool for neural network and dynamical systems simulations. It is C-based and provides a simple to use graphical interface as well as a core of routines needed in many applications. Routines required in special applications can easily be added. Felix is best suited for one and two-dimensional network models, but other topologies are possible as well.

1.2 The main philosophy of Felix

Main philosophy of Felix is to consider a neural network or more general dynamical system as a set of variables, x, which obey a certain dynamics, and a second set of parameters, p, which control this dynamics. Canonical examples are difference schemes, $x(t+1) = f(x(t);p)$ or differential equations, $dx/dt = f(x;p)$, which are omni-present in neural network and dynamical systems theory.

The Felix core implements and solves these dynamical equations and the graphical interface then presents the variables in various possible views like graphs and raster plots over time, images or functions displayed per single time-step, or xy-plots as on an oscilloscope.

The parameters of a simulation are further displayed as a collection of buttons and sliders in the graphical user interface, whereby it becomes possible to change them while the simulation is running and immediately observe the induced changes in the system dynamics. Figure 1.1 displays the graphical user interface of a typical small Felix program (actually a network of so-called integrate-and-fire neurons).



index 0 (0 – 24)
Min -0.01 Max 1.01
50 (0 – 100)

Min -0.01 Max 1.01
50 (0 – 100)

x: 0 y: 0 v: 0.924028

noise 5 (0 – 100)
input 100 (0 – 200)
coupling 60 (0 – 200)
Steps 50 (1 – 100)
Display Steps 1 (1 – 100)

Step : 1285

Figure 1.1: A typical Felix simulation showing a panel with control parameters at the bottom and windows for displaying variables of a running simulation at the top. Changing parameters is immediately reflected in the displayed variables.

A second design principle of Felix is that it aims at either "pool" networks comprising (more or less) large ensembles of potentially all-to-all connected units, or at layered one- and two-dimensional networks with a neighbourhood topology. Several such "pools" or "layers" may be combined into larger super-networks, see Figure 1.2. The first type of network model appears if local ensembles of cells in the brain are considered, the second if the focus is on the distributed processing within whole brain areas. In more general dynamical systems the first alternative refers to globally connected systems, whereas the second turns up, e.g., in partial differential equations and integro-differential equations. The core of the Felix simulation tool provides a number of often used routines to implement and simulate neural structures of the respective architecture, i.e., randomly connected pools, associative memories, or distributed systems with Gaussian or DOG (diference of Gaussian) lateral coupling kernels.

Recently Felix has been extended towards supporting various kinds of code parallelisation. Philosophy here is to simplify the development of parallel code for the types of networks described above

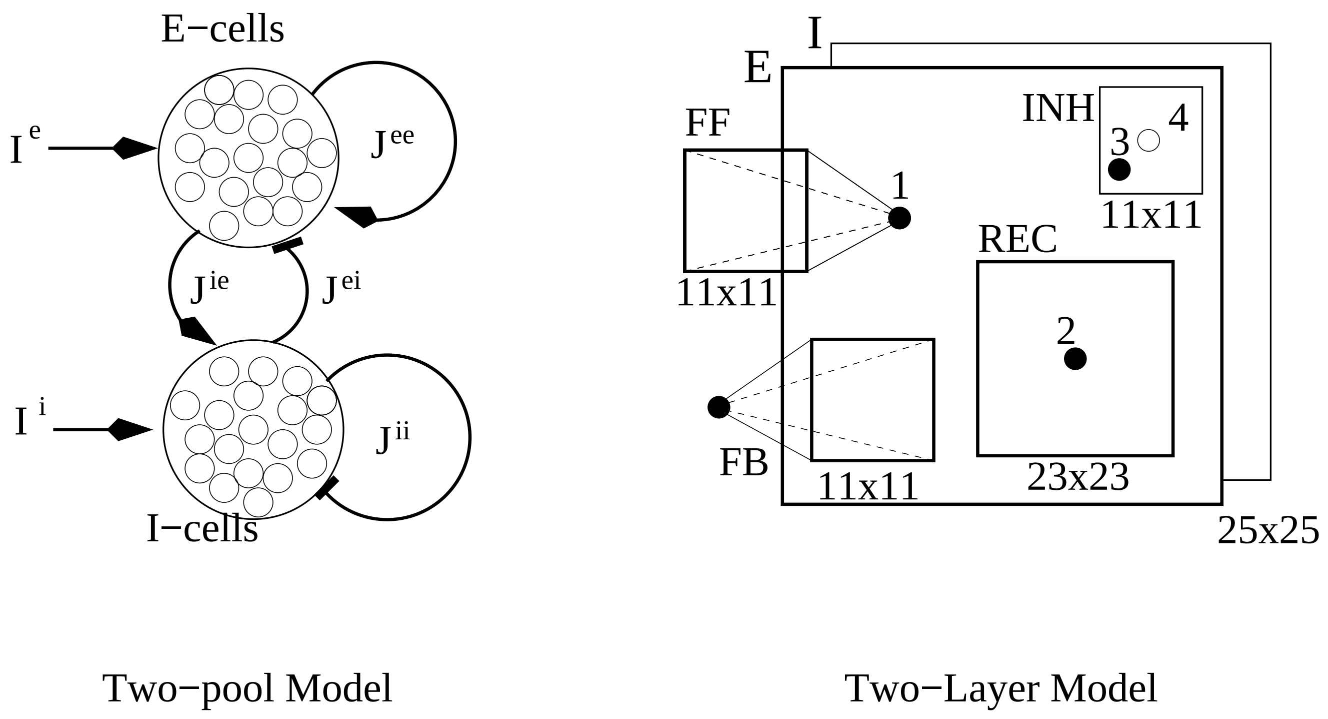


Figure 1.2: Left: A typical pool model (Wilson-Cowan Oscillator). Right: A two-layer, excitatory-inhibitory topographic neural field.

as much as possible. Using about a handful of simple constructs it is now in fact possible to write Felix programs that can be compiled on single CPU machines, where they reveal a graphical user interface, but that run also on Beowulf computer clusters. Small programs can therefore run on a PC or laptop, where the GUI and real-time simulation control nicely support an understanding of what is going on in the simulation. The same simulation, however, can now be easily scaled up and run on a much larger scale on a computer cluster without no or only small changes at the source code.

1.3 A little Felix History

Felix is old. The original program was written about 1990/91 in “multiC”, a dialect of C for the parallel computer “Wavetracer”, which (in the version we had available at that time at the University of Ulm, Germany) consisted of 4096 one-bit processors running at 8MHz in a SIMD-architecture (single instruction multiple data – each processor does the same on possibly different data). Each processor had something like 16MBit local memory and the processor grid was freely configurable as a 1, 2 or 3-D array. The early Felix was meant to serve as a graphical interface for that machine. The Wavetracer was about 20 times faster than a standard Sun-Workstation 15 years ago. When standard workstations became quicker, and in particular quicker than the Wavetracer, I ported Felix to the SunOs and Solaris operating systems, and later, when I discovered that even cheap laptops are faster than standard Sun-workstations, I further ported it to Linux. Now, I am almost exclusively using it under Linux on desktops, laptops, and computer clusters.

Because Felix is old it makes use of an outdated windows toolkit called XView. For a while that was standard for Sun X11 applications with the Open-Look look and feel. However, Sun stopped developing XView further in about 1995. Meanwhile it has been replaced by more modern toolkits like Motif, QT, and other packages. Although I often thought I should, I never found the time to recode the GUI using a modern toolkit. XView is still available and comes with some Linux distributions. It might however be that it is not installed on your machine by default. I am not sure it is available in 64 bit at the moment. You don't need the graphics libraries if you want to use the tool on computer clusters. Graphical interfaces don't make too much sense in high performance computing.

Some resources:

- Open-Look FAQ: http://www.faqs.org/faqs/open-look/01-general/
- XView FAQ: http://www.faqs.org/faqs/open-look/03-xview/
- O'Reilly provides free books about XView programming on their homepage http://www.oreilly.com/openbook/openlook
- Dr Andreas Knoblauch, a former collegue at the University of Ulm (now at Honda Research, Offenbach, Germany) has written C++ extensions for Felix which you can find here: http://www.informatik.uni-ulm.de/ni/mitarbeiter/AKnoblauch.html

Since relatively recently I am experimenting with parallelised Felix versions. This means Felix gets back to its roots, to parallel computers. The code contained in the distributed Felix version should be considered preliminary and is not well tested. However, it supports hybrid OpenMP/MPI code, which can be very useful for some types of layered network models of the brain. We are studying such models at the University of Plymouth as part of two big research projects: The EU-integrated FACETS project (comprising more than 100 scientists) and the UK-wide COLAMN projects (ca 10 research groups).

1.4 Installation Notes

Throughout this Guide it will be assumed that a functioning serial Felix evironment with graphical user interface is available. Only few sections in addition assume a parallel installation, in particular chapter 6. Appendix A explains, how Felix can be installed on serial and parallel computers, and computer clusters.

Chapter 2

Getting Started

This section presents the main features of Felix by showing a simple example and how it is implemented. The example will consist of a small network of leaky-integrate-and-fire neurons. It will demonstrate how a typical Felix program is structured, how a simulation can be controled by the graphical user interface, and how the simulated data can be conveniently written to files on disc.

2.1 General Program Structure

A Felix application consists of a single C-file. Each application needs to define five subroutines that define the GUI, the output of some data to files, a main-initialisation routine which is called once at start up, an initialisation routine which is called each time a simulation is reset, and a step-routine which contains everything to do in a single simulation step. Some or all of these functions can be empty. The smallest Felix program hence reads:

```
// The most simple Felix program

# include <felix.h>

NO_DISPLAY
NO_OUTPUT
main_init(){}
init(){}
step(){}
```

$< felix.h >$ is the main Felix header file that always has to be included and by itself includes several other header-files neccessary for proper compilation.

The macro `NO_DISPLAY` in the example actually expands to MakeDisplay(){}, e.g., an empty declaration of the graphical user interface. Similarly, the macro `NO_Output` likewise expands to MakeDisplay(){}, an empty declaration of output to files. Simple examples for the GUI and file output follow below. The GUI is treated in detail in chapter 3 and File Output in chapter 5.

The *main_init()*-routine contains initialisations needed only once during execution of a simulation program. It is executed when the program starts. It may load data from files or settings of

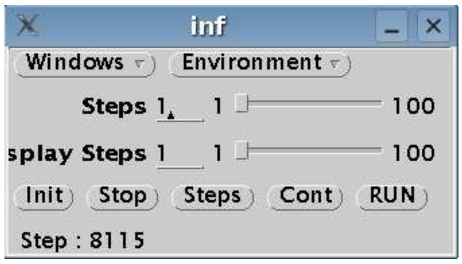


Figure 2.1: Graphical user interface generated by the minimal Felix program given in section 2.1.

parameter-values not accessible by sliders. If the application uses dynamically allocated vectors or arrays, memory for these variables MUST be allocated in main_init(), too, in particular if the variables are supposed to be displayed in the GUI.

The *init()*-routine in contrast is invoked each time a simulation is reset. The GUI provides init- and run-buttons in the main-window to do this. It typically contains code to initialise variables randomly. In conjunction with an additional counter variable in the code that increments each time a reset is performed the init-routine can also be used to scan a parameter range systematically and intialise each simulation in a well defined state using that counter.

The *step()*-function contains all things to be executed in a single simulation step. There is no constraint about the content of this function, but in genral it will comprise functions to iterate the dynamics of the simulated systems and possibly also to do some data analysis. The step()-function is repeatedly called if a simulation is in run-mode as long as it is not explicitely stopped. The GUI further supports single- and multi-step modes, in which case the step-routine is executed once or a fixed number of times.

The above trivial Felix program can already be compiled and executed. For that the code has to be stored in a C-file, i.e., a file called <sim_name>.c, where <sim_name> is some basename (e.g., "empty", because all subroutines are empty functions). Calling "Felix <sim_name>" (i.e., "Felix empty") compiles the program and generates an executable called <sim_name> ("empty"), which can be run from the command line. This should pop up the main-window of the simulation, which should look as displayed in Figure 2.1. (Note: The Felix example directory should contain an "empty.c" function, as well as others.)

The graphical user interface in Figure 2.1 contains simulation control elements that by default appear automatically in the GUI of each simulation program. The top label bar reflects the (base-)name of the compiled program. The "Windows-"button in general comprises a list of define dwindows, but for our simple example this list is empty. The "Environment-"button in contrast contains several entries (not shown) that allow to store and load parameter settings for the sliders below. The "Steps" and "Display-Steps" sliders control the multi-step and display mode of the GUI, respectively. If "Display-Steps" differs from 1, the variable windows (none is shown since they are empty, but see later) are updated only at the respective interval. This is useful to compress time in the display if the simulation step-size is small; it can sometimes also help to speed up simulations, because updating the display needs some time. The "Steps"-slider in the GUI cooperates with the Step-button just below. If the simulation is in multi-step mode, the "Steps"-slider specifies how many steps are executed until the simulation stops again, after the "Steps"-button has been pressed. This means, the bottom-row buttons control the overall execution of a simulation: Each time the Init-button is pressed the init()-routine is called. "Run" also calls the init()-routine, but afterwards the step()-routine iteratively – this is the standard simulation mode. “Stop” stops a simulation, “Step” runs a certain number of steps as explained above, and “Cont” (continue) enters the standard run mode again after a simulation had been stopped. Finally, the footer of the GUI main window contains a counter of the simulation step.

2.2 Example: Leaky-integrate-and-fire Neural Network

We now consider a more interesting example that indeed simulates something. This is a neural network comprising a certain number $N = 100$ of noisy leaky-integrate-and-fire neurons coupled randomly in a network. These simple neurons are described by membrane potentials x_i that integrate incoming input as low-pass filters with time-constant τ. If a potential crosses a firing threshold of 1 from below it is reset to zero and a spike is emitted. Spikes are represented binary by a second array of variables, z_i, $i = 1, \ldots, N$. Equation (2.1) describes the membrane dynamics and (2.2) the resest at threshold crossing:

$$\tau \frac{dx(t)}{dt} = -x(t) + I + \frac{J_0}{N} \sum_{i=1}^{N} J_{ij} z_j(t) + \sigma \eta_j(t) \tag{2.1}$$

$$\text{if} \quad x_i(t) \geq 1 \quad \text{then} \quad z_i(t) = 1, \text{ and } x_i(t) = 0 \quad \text{else} \quad z_i(t) = 0. \tag{2.2}$$

$\tau = 10$ in (2.1) is the membrane time constant and $J_0 = 1.1$ sets the coupling strength between units globally. The J_{ij} in contrast are individual couplings/synapses betwene pairs of neurons. In the simulation they are independent and identically distributed (i.i.d) Gaussian random numbers with mean 1 and standard deviation 0.4. The $`eta_i(t)$ in (2.1) are furthermore i.i.d. temporally Gaussian white noise processes with mean 0 and standard deviation 1. The factor σ scales this “noise” injected into the individual cells.

Networks of this type have been intensively studied in Neural Network Theory.

The following code implements the network model:

```
/* Example-program: inf.c  */

# include <felix.h>

# define N   100    /* number of neurons      */
# define tau  10.   /* membrane time constant */

float I = 1.1,      /* Common input to units  */
      J0 = 1.1,     /* Coupling strength      */
      sigma = .1;   /* noise level            */

Vector  x;          /* potentials             */
Matrix  J;          /* connections            */
bVector z;          /* vector of spikes       */
Vector  v;          /* auxiliary variable     */

NO_DISPLAY
NO_OUTPUT
```

```
int main_init()
{
  /* init. random number generator and stepsize */
  randomize( time(NULL) );
  SET_STEPSIZE( .1 )

  /* allocate vectors and matrices */
  J = Get_Matrix( N, N );
  x = Get_Vector( N );
  z = Get_bVector( N );
  v = Get_Vector( N );
}

int init()
{
  int i;

  Clear_bVector(N,z);
  Clear_Vector(N,v);

  /* init. potentials with random values between 0 and 1 */
  for (i=0; i<N; i++)
    x[i] = equal_noise();

  /* init. J with gaussian distr. random numbers */
  Make_Matrix( N, N, J, 1.0/N, .4/N  );
}

int step()
{
  int i;

  for (i=0;i<N;i++)  // leaky integration for all neurons
    leaky_integrate ( tau, x[i],
                 I + J0*v[i] + sigma*gauss_noise() );

  Fire_Reset( N, x, 1.0, 0.0, z );  // firing and reset

  bMult( N, N, J, z, v ); // redistribution spikes
}
```

Observe the general structure of the code. First felix.h is included as well as (some) parameters of the model are defined as macros (could also be variables). Then arrays for the neural variables x, J, z and an auxilliary array v are declared. The code still does define an empty GUI and data output routine (NO_DISPLAY and NO_OUTPUT). After that the three obligatory functions main_init(), init(), and step() follow.

main_init() initialises the random number generator and sets the simulation time-step to 0.1. Afterwards the routine allocates the three vectors v, x, z, and the array J. Note that the z-array is a “bVector” – a *binary* Vector. [Many functions in Felix operate either on floating point vectors and matrices or on binary ones, where binary values (0/1) are represented by the C-type “char”.]

The init()-funtion initialises the data-arrays: v and z are cleared, ie., set to 0; the potentials are set to equally distributed random numbers in the range [0,1[; and the coupling matrix J is filled with i.i.d. Gaussian random numbers, N(1., 0.4).

Finally, the step()-routine implements the dynamics of the network. It mainly uses functions from the Felix libraries. The leaky integration in equation (2.1) is coded explicitely using the macro “leaky_integrate”, which implements a simple Euler-scheme to integrate the low-pass dynamics. “Fire_Reset()” afterwards does the thresholding part of the leaky-integrate-and-fire dynamics, and “bMult()” computes the Matrix-Vector product between the coupling Matrix J and the binary vector of spikes z. The result v is used in the leaky integration in the next step.

Again the code shown can be compiled and run using Felix, but since it neither defines graphical nor file-output, we would not be able to observe what the network is doing. The interface would look as in Figure 2.1. Therefore, we next add some graphical output.

2.3 Adding a Graphical User Interface

The graphical user interface serves different tasks, the two most important are displaying variables of the simluation and providing sliders to control it (other task concern file I/O and saving/loading parameters). In the first case the information flow is from the running simulation to the GUI, whereas in the second it is the other way round – the user changes sliders, which in turn modify simulation parameters. The next two sub-sections explain how these tasks are set up.

In general the function MakeDisplay() represents the main-interface between the C-code and the XWindows-System. It contains statements that define how variables shall be displayed on the screen and, if needed, declares buttons (called switches) and sliders, which allow for interactive control of a running simulation. MakeDisplay always generates a main-window with several buttons and sliders, which are used to control the simulator-kernel even if the MakeDisplay() function is explicitly declared empty or the macros NO_DISPLAY is used (which does the same), see Figure 2.1.

2.3.1 Displaying Views onto Variables

As outlined in section 1.2 a simulation can be considered a dynamical system comprising variables and parameters. Variables are displayed to the user and parameters can be odified by it. The code below shows how a typical Graphical User Interface for the leaky-integrate-and-fire neural network program could be declared.

```
BEGIN_DISPLAY

  WINDOW("time courses")

    IMAGE( "x", AR, AC, x, VECTOR,  10, 10, 0.0, 1.0, 4)
    RASTER( "x", NR, AC, x, VECTOR,  N, 0, 0.0, 1.0, 1)
    GRAPH( "x", NR, AC, x, VECTOR,  N, 0, 0, 0, -.01, 1.01 )
    RASTER( "z", NR, AC, z, bVECTOR, N, 0, -.01, 1.01, 2)

  WINDOW("couplings")
```

```
    IMAGE( "J", AR, AC, J, CONSTANT MATRIX, N, N, -4./N, 4./N, 4)

END_DISPLAY
```

The macros BEGIN_DISPLAY and END_DISPLAY enclose the definition of a GUI; they expand into a MakeDisplay(){} function body (thus, you can also define this function directly wthout using the macros). Everything between the two macros is executed when the GUI is build. In the present example two windows are defined with names “time courses” and “couplings”, respectively. It is possible to define an arbitrary number of such display windows.

Each display window can in turn comprise an arbitrary number of so-called “views”. A view is a view of a variable, e.g., a scalar, vector or a matrix. Each variable can be viewed in different ways, and section 3 describes the possibilities in detail. Here, it may sufice to observe that the window “couplings” displays the $N \times N$ coupling matrix J as an IMAGE, i.e., a grey-scale coded image that reflects the values of the matric entries. Because J is declared as a CONSTANT MATRIX in the IMAGE-definition, the image of J is updated only once, after each call to the init()-function. This saves unnecessary updates, which cost time.

On the other hand, the window “time courses” defines four views, three different onto the x-variables (potentials), and one on the spikes z. The potentials are displayed as an IMAGE of size 10 (just for demonstration), a RASTER which displays the potentials over times as a grey-level plot, and a GRAPH, which selects a single potential trace and plots it as a function over time. The spikes are, finally, also plottet as a RASTER, ie., the values of the whole array are displayed over time. More about this later, when we look at the actual graphical output (Figure 2.3 for the impatient).

2.3.2 Coupling of Parameters and Panel Controls

The views on variables defined in the previous section allow to observe in real time variables of the simulations. However, we might also want to change parameters and see where that leads to. To do this we have to add control elements to the main window of the GUI (the on we laready knwo from Figure 2.1). These elements can then be coupled to parameters of the simulations.

There are two types of control elements available in Felix, Switches and Sliders. Switches are represented by buttons, they can be ON or OFF, and thereby they can “switch” code execution between alternative segments (Switches are not used in this section, but see section 3). The second control element are Sliders. These can take values in a whole range and can thereby represent continuous parameters of the simulations.

How does this work inpractice? Let us assume we want to control the parameters I, J_0 and σ in the simulation of the leaky-integrate-and-fire network. These are the global input, the global effective coupling strength, and the noise level. For each of these we have to define a new variable of type SliderValue (the reason for this follows soon). These new variables we have to embed in the GUI, and we can use them in the simulation code as well. The code below shows how this is achieved.

```
SliderValue sI     = 100;
SliderValue sJ0    = 50;
SliderValue ssigma = 0;

BEGIN_DISPLAY

  SLIDER( "input",     sI, 0, 200)
  SLIDER( "coupling", sJ0, 0, 200)
  SLIDER( "noise", ssigma, 0, 100)

  WINDOW("time courses")

  ....

END_DISPLAY
```

sI, sJ0 and ssigma are the new variables of type SliderValue. The SLIDER()-macros then add the slider to the GUI, giving them names and certain lower and upper bounds. This code-snippet reflects one problem with Sliders: Constrained by the XWindows/XView system they can only take integer values. In the example these ranges are $\{0, 1, 2, \ldots, 200\}$ for input sI and coupling $sJ0$, and $\{0, 1, 2, \ldots, 100\}$ for the noise level $ssigma$. Accordingly, when the variables are used in the code implementing the dynamic equations of the simulated system they have ot be scaled appropriately. For instance in the step()-routine we could have code like

```
  for (i=0;i<N;i++)  // leaky integration for all neurons
    leaky_integrate ( tau, pot[i],
                 0.01*( sI + sJ0*v1[i] + ssigma*gauss_noise() ) );
```

The factor 0.01 scales the ranges for sI , sJ0 into the intervals [0,2[and that for ssigma into the range [0,1[. This is somewhat uncomfortable, but one gets used to it quickly.

Fianlly, note that now that we have replaced the original variables I, , $J0$ and σ by slider variables, we can delete their original declarations in the program. They don't appear in the code anymore, but instead they are controled by the graphical user interface, see Figure 2.2.

2.3.3 Running simulations using the graphical interface

Figure 2.2 displays the GUI after the control elements have been added. The display windows we have defined are still hidden. We can open them by right-clicking in the "Windows"-button, which pops up a list of the availabe windows.

Figure 2.3 shows the interface after the display windows have been opened and placed on the screen. On top of the control panel is shown the coupling matrix and to the left of both the window containing the variable "time courses".

By left-clicking the "Environment"-button this configuration can be saved such that the GUI comes up in the same state the next time the program is started (right clicking the "Environment"-button gives some more options). This automatic loading of parameters from a default environment file overrides any explicit initialisations of slider variables possibly done in the source code.

The Felix example directory contains the code of a leaky-integrate-and-fire network with GUI and file output. You might want to experiment with it, before proceeding to the next section which

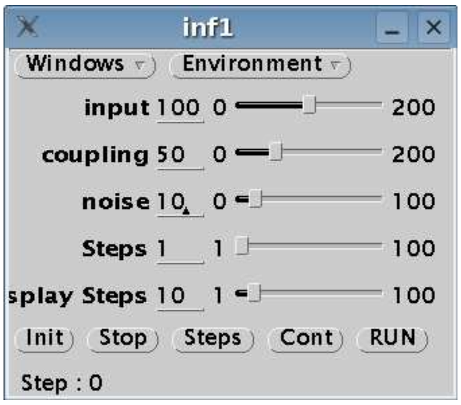


Figure 2.2: Graphical user interface after adding sliders for parameters of the simulation.

describes how file output is declared. In particular note, that the label on top of each view is clickable and brings up control panels for the grey-scales of images and rasters, or the selected variable index in graphs that display arrays.

2.4 Adding Output of Data

Analogous to MakeDisplay() which defines the graphical output, the function MakeOutput() defines output that is supposed to go to files. The macro NO_OUTPUT can be used if no such output is needed. The macros BEGIN_OUTPUT and END_OUTPUT in turn enclose code for data output. This defines a function MakeOutput() which is called before the first simulation step and after each subsequent one. It is possible to select sub-ranges for output, which is explained in detail in chapter 3. The following code shows how variables of the leaky-integrate-and-fire model can be saved.

```
BEGIN_OUTPUT

  OUTFILE("potentials")
    SAVE_VARIABLE( "pot", x, VECTOR, N, 0, 0, 0, 0 )

  OUTFILE("spikes")
    SET_SAVE_FILE_FLAG( THISFILE, ASCII, ON )
    SAVE_VARIABLE( "out", z,  bVECTOR, N, 0, 0, 0, 0 )

END_OUTPUT
```

Two output files are defined "potentials" and "spikes" with obvious meaning. An arbitrary number (up to operating system constraints) can be open, each of which can save a number of variables per step. Those are stored in sequential order, which might cause problems when the data has to be reread, because of the possibly complicated record structure. It is in probably usually more convenient to store only one variable per files as in the shown example. The variable x is stored as a vector of length N to the file "potentials", wheras z is stored as a binary vector of length N to file "spikes".

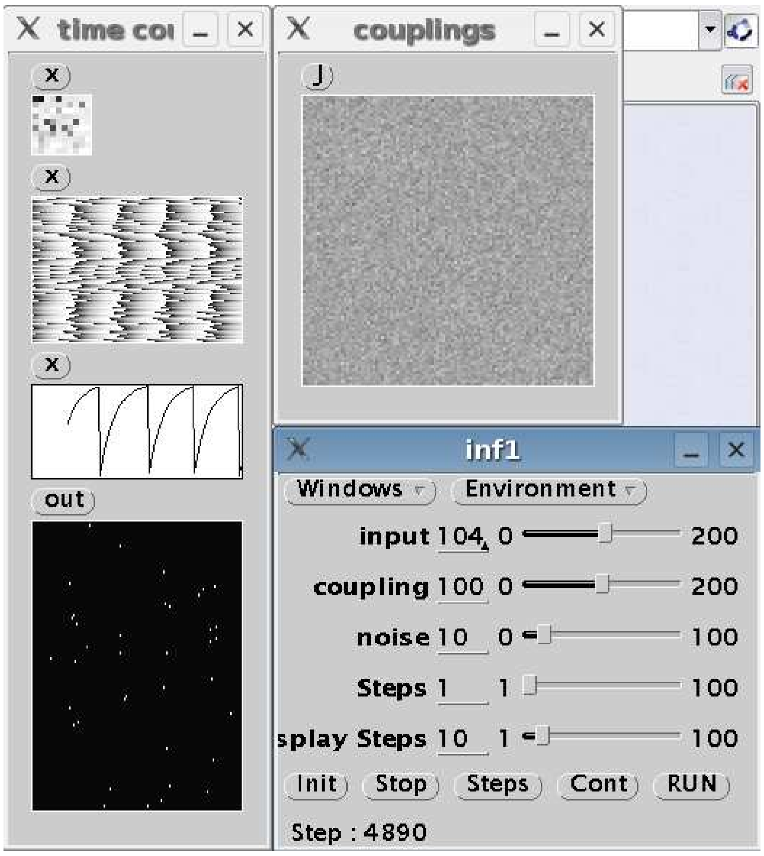


Figure 2.3: User interface showing the control panel and the two windows created for displaying dynamic variables.

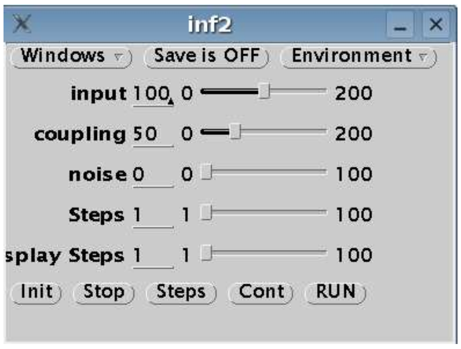


Figure 2.4: Control panel of the graphical user interface after output files have been declared in the program.

By default data is saved in binary format, so that it would not be readable by humans, but save space. The default is changed for the second file in the example, where a flag is set for (human-readable) ASCII output.

Figure 2.4 shows the main GUI window after non-empty file output has been defined. The new button "Save is OFF" indicates that the ouput has not yet been activated. If it is pressed, file output starts. If it is pressed repeatedly during a simulation, the stat of the button toggles, and data generated during the activated phases are appended to the output files. Right-clicking on the "Save is OFF"-button brings up further options, not all of which are fully implemented nor well tested.

Have a look into chapter 5 (or the example programs, or the source code of output.c/h) for possibilities to select sub-ranges of array variables for output. This can be useful for large simulations because otherwise the amount of generated data can quickly become tremendeous.

Chapter 3

Graphical User Interface

Chapter 2 presented a brief example of how Felix programs are structured and what the main properties of the Graphical User Interface are. The present chapter looks into the GUI in more detail.

In general each Felix program has a main window with control elements for running a simulation and manipulating its parameters in real-time. Switches are binary (ON/OFF) controls that allow for a conditioned execution of code segments. Sliders in contrast can take values in a range of numbers and can therefore be used to set parameters of a simulation. Beside the main window a Felix simulation in general will have one or more display windows. These can contain graphics objects of various types, which display views on variables as, e.g., graphs, functions, images, or plots. Finally each Felix simulation has an "Environment" allowing the user to store and load parameter settings in external files. The present chapter will go through the mentioned components step by step.

3.1 Creating a GUI

Each Felix application has to define which objects (variables, vectors, matrices ...) are displayed on the screen and how this shall be done. For this a function

```
    void MakeDisplay(){...}
```

has to be supplied which contains definitions of the graphics objects to be displayed. The function-body of "MakeDisplay" can be empty, if no graphical output is needed. In that case a basic main window is still generated, see Figure 2.1, but no display windows. There are three Macros that support the definition of the interface

```
#define BEGIN_DISPLAY  void MakeDisplay(){
#define END_DISPLAY    }
#define NO_DISPLAY     void MakeDisplay(){}
```

Beside a number of buttons to initialialise, run, stop, and resume a simulation, the each GUI by default contains two sliders "Steps" and "Display Steps". These control the display and multi-step mode of a simulation. “Display Steps” sets the interval in simuation steps at which the graphics objects in the display windows are updated. “Steps” in contrast sets the number of steps that are executed in multi-step mode (i.e., after stopping an initialised simulation) when the Steps-button of the GUI is pressed. The maximum steps of both these sliders by default is 100, which is convenient for most situation. Should it be necessary, the numbers can be changed using the macros MAXSTEPS() and MAXDISPLAYSTEPS() in the definition of MakeDisplay().

```
  MAXSTEPS( steps )
  MAXDISPLAYSTEPS( steps )
```

In very new versions of Felix the colormap for the variable views can be selected by using the macro `COLOR_MAP( map )` somewhere at the top of the display declaration. Possible values for “map” are:

CMAP_BW : The default gray-scale map; black: low-values; white: high values

CMAP_RED : Black to crimson-red intensity coded (quite hellish)

CMAP_GREEN : All green (looks like the aliens are around the block)

CMAP_BLUE : All blue (deep not light blue)

CMAP_RAINBOW : Blue - green - red colour scale (quite fancy)

In the non-gray maps the lowest and highest values are black and white, respectively. This makes clipping at range boundaries nicely visible.

3.2 Simulation Control Elements

3.2.1 Switches

Switches are logical flags, that may be used in a simulation to interactively select execution of different parts of the source-code.

A switch must be globally declared as a variable of type ’SwitchValue’ on the top of the application-source file.

A switch can be ON or OFF:

```
#define OFF 0
#define ON  1
```

To create a button in the main-window that affects the switch-variable the function MakeSwitch() or the Macro SWITCH() must be called in MakeDisplay() :

```
#define SWITCH(name, var) MakeSwitch(name, &var);
```

Here “name” is a string that appears on the switch-button in the GUI and “var” is its accociated variable of type SwitchValue.

If a user wants to change values of switch-variables at the source-code level the function SetSwitch() or the Macro SET_SWITCH() MUST be used. Simply assigning a value to a switch-variable is not enough, because the new value will not be signalled to the XWindows-system, such that the state of the switch would be no longer represented by its corresponding button.

```
#define SET_SWITCH(var, value) SetSwitch(&var, value);
```

“var” is the variable to be set; possible values are ON or OFF.

Example:

```
...

SwitchValue sw = OFF;         /* define the switch-variable */
...

BEGIN_DISPLAY
...
SWITCH( "this-or-that", sw)  /* define the switch-button    */
...
END_DISPLAY



int void step()

  ....

  if (sw)                     /* execute code depending on
                                 the state of sw.           */
  {     /* do this */
    ...
  }
  else
  {     /* do that */
    ....
  }

}
```

3.2.2 Sliders

As switches sliders are used to interactively control a running simulation. The difference is, that they are multi-valued and thus may be chosen to influence parameters of the model. To create a slider the user has to declare a global variable of type 'SliderValue' (i.e., int). This variable is associated with a graphical slider in the GUI by a call to the function MakeSlider() or the macro Slider() inside the initialization routine MakeDisplay().

Sliders appear in the main-window in the order of their declaration in MakeWindow(). The macro SLIDER_COLUMNS(columns) can be used to arrange them in more than 1 column (default).

```
#define SLIDER(name, var, min, max)  MakeSlider(name, &var, min, max);
```

"name" is a string that appears to the left of the slider. "var" is the variable of type SliderValue that stores the current value of the slider. "min" and "max" set the range allowed for changes in the sliders value.

To set or change slider-values at source-code level the function SetSlider() (or macro SET_SLIDER) must be used:

```
#define SET_SLIDER(var, value) SetSlider(&var, value);
```

"var" is the name of the slider-variable and "value" the new slider-value of type SliderValue (i.e., int).

Unfortunately, XView restricts sliders to integer-values. Thus, if an application contains floating-point parameters which shall be modifiable from the graphical interface, one has to scale the corresponding slider-values to the appropriate range.

Example :

```
SliderValue param = 50;    /* define and initialize a slider-variable */
...


BEGIN_DISPLAY
...
SLIDER( "parameter", param, 0, 100)  /* generate an instance of a
                                        XView-slider in the main window
                                        associated with variable "param"
                                        The name of the slider is
                                        "parameter", its range [0,100]
                                      */
...
END_DISPLAY



int void step()
{
  float float_param;
  ..

  float_param = .01*param;      /* this casts the slider-value to float
                                 * in the range [0,1.0]. Observe, that
                                 * only 100 different values are possible!
                                 */

  ..

  if (any_condition)
    SET_SLIDER( param, 50 )     /* this sets the slider to a well-defined
                                 * value (here 50).
                                 */

}
```

3.2.3 Timer

Usage of the Macro TIMER() in the definition of MakeWindow() creates an extra slider which influences the time between two successive simulation-steps.

```
TIMER( max )
```

The timer slider will have a range from 0 to max. If the value is zero the timer is off. Otherwise it effects the times between calls to the step() routine in a running simulation. The value in principle is supposed to be in Milliseconds, but this shouldn't be taken too seriously.

3.3 Display Windows and Views on Variables

3.3.1 Display Windows

The Macro WINDOW() or function MakeWindow() create a new window for graphical output. The string "name" appears at the top of the window and in the Window list of the main control window.

```
#define WINDOW(name) MakeWindow(name);
```

The WINDOW-statement must be called in MakeDisplay() before any other output can be directed to the screen, i.e. before any graph, image, raster, or other variable views are defined.

Several windows can be defined by repeated calls to WINDOW(). In this case the last declared window is always the active one, meaning that subsequently declared graphics objects are placed into that window.

All window-names are collected into the "Windows"-menue at the top left of the main control window. If a window is closed selecting it from this menue will reopen it.

3.3.2 Views

After a Window has been defined it can be filled with graphical views on simulation variables (images, graphs, etc). The functions to create the various possible views all have a similar structure. Consider, e.g., creation of an image by using the macro IMAGE():

```
IMAGE(name, row, col, var, type, dim_x, dim_y, min, max, zoom)
```

The first argument is the “name” of the view. It will appear on a button on top of the view.

“row” and “col” are two arguments to control positioning of the view in the display window. This is covered in the next subsection 3.3.3.

“var, type, dim_x, dim_y” then characterise what is actually displayed. The type of a displayed variable “var” must be declared and, if it is a vector or an array, also its dimensions. Possible display types are described below in subsection 3.3.4.

The last argument of a view definition, “zoom”, is a (small) integer number that controls how big a view appears on screen. Default value is 1. In that case, e.g., each entry of a matrix-valued variable will be displayed by one pixel in a rectangular image. Larger numbers for zoom correspond with more pixels and bigger images.

3.3.3 Placement of Views inside a Window

There is a simple mechanism to control positioning of view elements.

The 2cd and 3rd arguments of a view-definition are coordinates for the upper left corner of the view. These can be given directly by specifying raw pixel coordinates.

Alternatively, each display window can be considered as being partitioned into a coarser rectangular grid. Several macros support placing views in that coarse grid.

R0, AR and NR can be used as values for the 2cd, and C0, AC, and NC as values for the third argument of a view defining function.

R0 and C0 specify the first row and column, respectively.

AR and AC specify that the view has to be placed in the actual row or column, respectively.

NR and NC specify that the view has to be placed in the next row or column, respectively.

Example (cf., section 2.3)

```
BEGIN_DISPLAY

  WINDOW("time courses")

    IMAGE(  "x", AR, AC, x,  VECTOR, 10, 10,  0.0,  1.0, 4)
    RASTER( "x", NR, AC, x,  VECTOR,  N,  0,  0.0,  1.0, 1)
    GRAPH(  "x", NR, AC, x,  VECTOR,  N,  0, 0, 0, -.01, 1.01 )
    RASTER( "z", NR, AC, z, bVECTOR,  N,  0, -.01, 1.01, 2)

  WINDOW("couplings")

    IMAGE(  "J", AR, AC, J, CONSTANT MATRIX, N, N, -4./N, 4./N, 4)

END_DISPLAY
```

This example defines two windows with names “time courses” and “couplings”. The first window contains four graphics views, the second only 1. In both cases the first view is placed at position (AR, AC), the actual row and actual column, which by default after creating a new window with WINDOW() is equal to (R0, C0), the upper left location in the coarse grid. The subsequent views in the first window are then placed at (NR, AC), meaning the next row but actual column. Therefore, the four views are placed in a single vertical column. In contrast, replacing (NR, AC) by (AR, NC) in the code would place the views all in a horizontal row, and (NR, NC) places them along the diagonal of the coarse grid (which wouldn't look too nice).

3.3.4 Types of Display Variables

Felix supports displaying of variables of three base types: char, int, and float. A fourth type, packed bits, is obsolete and shouldn't be used. Displaying double, long, and unsigned variables is not supported.

Variables can be scalars or vectors / arrays. There are several type macros that can be used in the view display type definitions:

```
#define CHAR_TYPE       0x02
#define INT_TYPE        0x04
#define FLOAT_TYPE      0x08

#define ARRAY_TYPE      0x20

#define ARRAY_CHAR_TYPE     (ARRAY_TYPE | CHAR_TYPE)
#define ARRAY_INT_TYPE      (ARRAY_TYPE | INT_TYPE)
#define ARRAY_FLOAT_TYPE    (ARRAY_TYPE | FLOAT_TYPE)
```

The basic display types above are conveniently redefined in some of the Felix libraries, e.g.:

```
vector.c/h : VECTOR,  MATRIX  = ARRAY_FLOAT_TYPE
            bVECTOR, bMATRIX  = ARRAY_CHAR_TYPE
nn.c/h :      LAYER           = ARRAY_FLOAT_TYPE
           SPIKE_LAYER        = ARRAY_CHAR_TYPE
```

It is sometimes desired not to provide just a variable to a view, but a pointer to a variable. The variable the pointer references can then change dynamically in every display step. The POINTER-macro sets the respective type bit.

```
#define POINTER        0x8000
#define TO             |
#define CONST_BIT      0x4000
#define CONSTANT       CONST_BIT |
```

A variable can be declared CONSTANT if it does not need to be updated during a running simulation. A CONSTANT variable is updated only after a call of the init()-function at the beginning of a simulation, ie., by pressing the Init- or Run-buttons of the GUI.

Examples

```
BEGIN_DISPLAY

  WINDOW("time courses")

    IMAGE(  "v", AR, AC, v, MATRIX, 10, 10,  0.0,  1.0, 4)
    IMAGE(  "z", AR, NC, z, CONSTANT MATRIX, 10, 10,  0.0,  1.0, 4)
    IMAGE(  "y", AR, NC, y, POINTER TO bVECTOR, 10, 10,  0.0,  1.0, 4)
    IMAGE(  "x", AR, NC, x, POINTER TO CONSTANT VECTOR, 10, 10,  0.0,  1.0, 4)
    ...
```

The first IMAGE defines a view on a MATRIX (ARRAY_FLOAT_TYPE) of size 10×10. This is how a view definition usually declares a variable type. The second IMAGE also defines a view on a MATRIX , but because the matrix is declared CONSTANT the graphics view will only be updated when the simulation starts. The third IMAGE refers to a binary vector image (bVECTOR = ARRAY_CHAR_TYPE). However, a pointer to the variable y is declared so that the array y may change dynamically. The fourth image also declares the variable z a pointer, but this time a constant one, so that it could change where it points at, but its view is refreshed only at the beginning of a simulation. (I can't remember I ever used this possibility during the last 15 years).

3.3.5 Image-Views

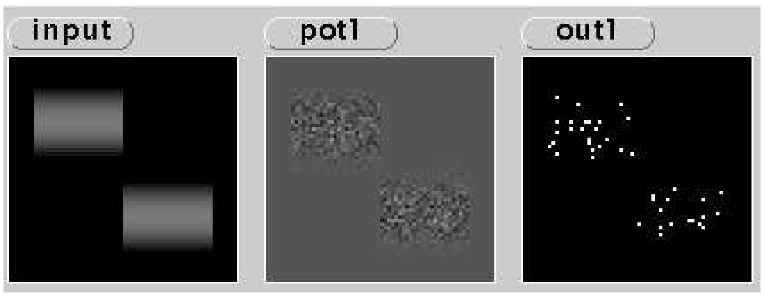


Figure 3.1: 2d grey-scale images of 3 arrays in a neural field model. Left: input; middle: potentials; right: spikes of the cells in the model.

An image is a two-dimensional grey-scale plot of the current state of a two-dimensional variable. It is created by calling the macro IMAGE() inside the function MakeDisplay():

```
IMAGE(name, row, col, var, type, dim_x, dim_y, min, max, zoom)
```

Here “name” is a string that appears on a button above the image. “row” and “column” give the position of the image in the currently active window, see section 3.3.3.

“var” is the variable to display as an image, “dim_x” and “dim_y” its dimensions, and “type” its display type as described in section 3.3.4.

The floating point variables “min” and “max” set the grey-scale of the image and should be set to the expected min- and max-values for the variable.

The integer valued argument “zoom” sets the size of the image. Each element of the variable is displayed as a square of size zoom×zoom.

Vectors, i.e. one dimensional arrays, can be displayed as images, too, by providing appropriate dimensions in the call to IMAGE(). If dim_x*dim_y is bigger than the actual vector size, the behaviour is undefined and core dumps can potentially occur. If it is smaller the remaining components are not displayed.

3.3.6 Raster Plots

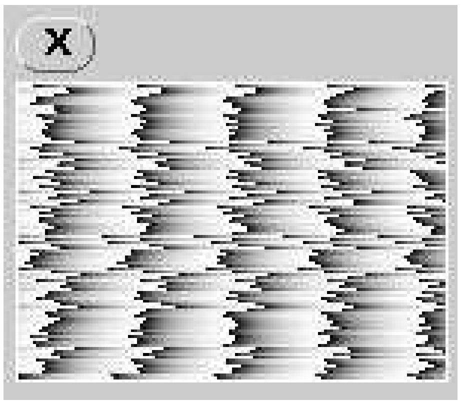


Figure 3.2: Raster plot of the grey-scale-coded potentials of 100 leaky-integrate-and-fire neurons over time.

A raster plot displays a vector or 2d-array as a function of time. Each component of the variable is shown grey-level coded on a seperate line.

```
RASTER(name, row, col, var, type, dim_x, dim_y, min, max, zoom)
```

The arguments are the same as in image. If “dim_y” is not zero it is assumed that “var” is a 2-dimensional array that has to be interpreted as a vector of length dim_x*dim_y.

“zoom” only sets the height of each displayed line (in pixels). It has no influence on the number of time-steps that fit on one line.

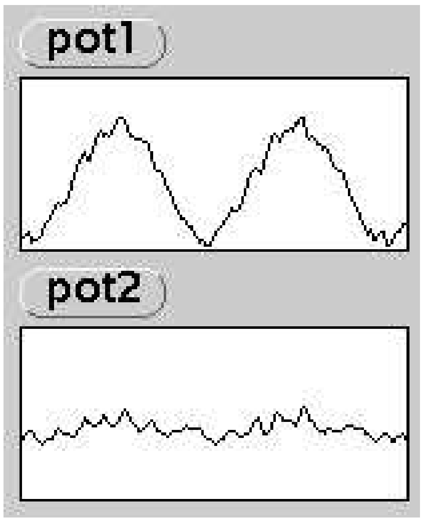


Figure 3.3: Two single variable graphs over time

3.3.7 Single Variable Graphs

A graph displays a single scalar variable, or a component of a 1 or 2d-array as a function of time.

```
GRAPH(name, row, col, var, type, dim_x, dim_y, x, y, min, max)
```

All but the "x" and "y" parameters are the same as in IMAGE.

In case of ARRAY_TYPES (see section 3.3.4) "x" and/or "y" specify which component of the variable has to be displayed initially.

3.3.8 xy-Plots

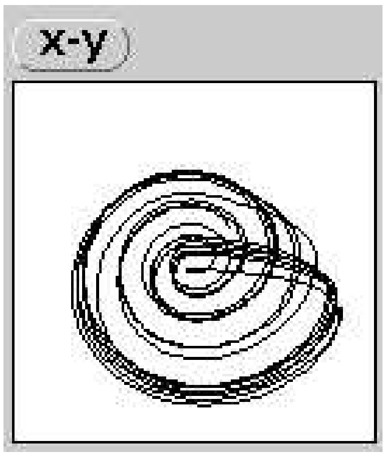


Figure 3.4: Example showing an x-y-plot of two variables of a Roessler oscillator

This view object displays an xy-plot of two variables. The variables may be independently chosen as single scalars or components of 1 or 2d-arrays.

```
PLOT(name, row, col, var1, type1, dim_x1, dim_y1, x1, y1, min1, max1,
                     var2, type2, dim_x2, dim_y2, x2, y2, min2, max2,
                     zoom)
```

All parameters are the same as in GRAPH, but note that one has to specifiy two variables with adjoined information about variable type, the subelement to select from arrays, and the grey-scale settings.

3.3.9 Arrays of Images

This type of view mainly aims at displaying 2-dimensional arrays of 2-dimensional images as they arise, e.g., in neural field models, where each local unit in a 2d-field has an individual 2d-lateral connection kernel. 1-dimensional arrays of 2-dimensional images (e.g., a stack of cortical layers) can also be displayed.

```
IMAGE_ARRAY(name,row,col, var,t, dim_x, dim_y, d_x, d_y, x, y, min,max,zoom)
```

"name, row, col, min, max, zoom" have the same meaning as usual (see IMAGE).

"var" and "t" define the variable and its display type (which must be an ARRAY_TYPE (usually MATRIX) and can be a POINTER type)

"dim_x" and "dim_y" define the dimensions of the array of images. If "dim_y" is zero, but "dim_x" positive a one dimensional array of images is assumed.

"x" and "y" specify which of the sub-images of the array of images is displayed initially (can be overridden by the Environment).

"d_x" and "d_y" define the size of the displayed images in x and y direction. They must both be positive.

3.3.10 Functions

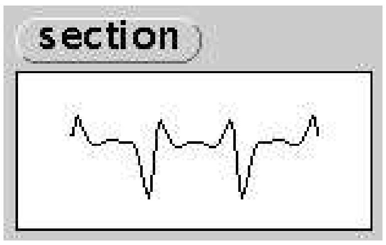


Figure 3.5: A function view

A function view plots a one-dimensional array (VECTOR) as a function of the array index. Single functions , and one- and two-dimensional arrays of functions are possible (cf., IMAGE_ARRAYS).

```
FUNCTION(name, row, col, var, type, points, dim_x, dim_y, x, y, min, max)
```

“name, row, col, var, type, min, max” are the same as in GRAPH.

“points” is the number of data points in any individual array that is to be plotted as a function.

“dim_x, dim_y, x, y” are used for arrays of functions. If “dim_x” is bigger than 0 and “dim_y” is zero a one-dimensional array of functions is assumed; if they are both bigger than 0, a two-dimensional one. “x” and “y” define the initially selected function to display (can be overridden by the Environment).

3.4 View Settings Frames

Each graphical view object in a window has a “settings frame” associated with it that can be used to control the grey-scale ranges of the view and to select sub-elements in case of array variables for graphs, functions, or image arrays. The settings frame pops up, if the button of a view showing the view’s name is pressed.

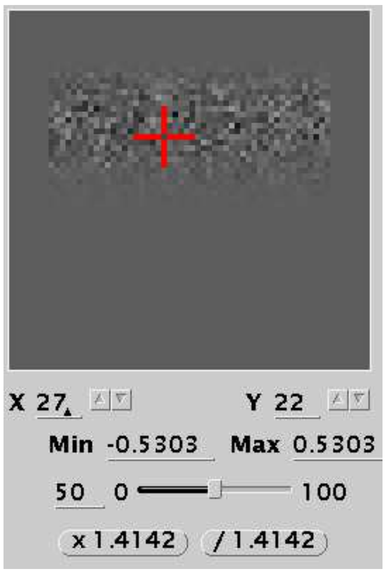


Figure 3.6: A two-dimensional settings frame

Figure 3.6 shows a two-dimensional settings frame as it could occur for a graph of single element of a 2d array. The frame shows the full array as a grey-scale image where the particular element to display is indicated by the red crosshair. The element can further be selected by the x and y textfields. If a variable has to be selected from a one-dimensional array only, the 2D-image is replaced by a slider. If the variable to display is a scalar, no extra element selectors will appear in the corresponding settings frames, but only the controls for setting the grey-level.

The definitions of all views contain arguments “min” and “max”. These set the initial grey-scale for that view. They can be changed in the settings frame, too. If the scale is changed, the new settings can be stored to an environment file (see section 3.5).

3.5 Loading and Saving GUI Settings

The Felix GUI for convenience provides the possibility to load and store settings of the graphical interface. The "Environment"-button on the main control window serves this task.

Right-clicking the "Environment"-button brings up a menue with four options.

Save This saves the current settings in a default file

Load This loads settings form the default file

Save as .. This pops up a window where the current settings can be stored in an arbitrary file

Load ..] This pops up a window where settings can be loaded from an arbitrary file

Left-clicking the "Environment"-button by default saves the current settings in the default file.

The default file is located in a sub-directory "env" of the current working directory, ie., the directory the executable is located and run in. The default file has the same name as the executable.

The default file does not exist until it is created (by left-clicking the "Environment"-button). If the environment directory "env" does not already exist, it is created, too.

If a default file exists it is automatically loaded when an application starts. This overrides any explicit initialisations of switch or slider values, image grey-scales, or sub-selections in views that can plot array objects. Occasionally this behaviour is unwanted,. You then need to rename or delete the default environment file in the "env"-subdirectory.

Note: Changing the number of graphic objects (switches, sliders, windows, views) in the GUI definition of an application typically invalidates the environment file(s). Instead of using the file, the application will print an error message on the screen. This is sometimes uncomfortable for complex applications, because all settings have to be set anew. It can then be easier to hand-edit the environment files: If proper entries for the new (or deleted) objects are added, the file can be used again.

Chapter 4

Libraries

Felix comes with a number of function packages / libraries suitable for tasks often encountered in the modeling of neural networks and dynamical systems. The present section provides an overview.

4.1 Outline: Pools and Fields

Although not restricted to them, two types of models have been in the main focus during the design of Felix - networks comprising homogeneuos neural pools and layered, topographically ordered neural fields, cf., Figure 1.2 in section 1.2.

Given a single neural pool of N neurons its dynamics could be described mathematically by

$$\tau\phi_i(t) = -\phi_i(t) + I_i(t) + \sum_{i=1}^{N} w_{ij} f(\phi_j(t)) + \sigma\eta_i(t) \tag{4.1}$$

Here, the cells are modelled by a single variable for their membrane potentials, $\phi_i,\ i = 1, \ldots, N$, and by a graded sigmoid output or rate-function f. Single units are identical: they obey the same mebrane low-pass dynamics with time-constant τ and have the same rate-function. They might, however, receive different inputs $I_i(t)$ and noise $\eta_i(t)$ of strength σ, and their synaptic weights $w_{ij},\ j = 1, \ldots, N$ will differ. More complicated single neuron models are of course possible. Cells, in general, also don't need to be identical.

The dynamics of a single neural field in contrast can be written as

$$\tau\phi(x,t) = -\phi(x,t) + I(x,t) + \int w(x,x')f(\phi(x,x',t)) + \sigma\eta(x,t) \tag{4.2}$$

In contrast to (4.1) cells do not just have indexes, but a continuous spatial location x (which will, of course, typically be discretised in computer models). Units at one location interact only with neighbours nearby. This is reflected by the synaptic kernels $w(x,x')$ in (4.2). Beside this, the meaning of the symbols in equations (4.1) and (4.2) are the same.

Both kinds of models need similar construct to define and simulate the single units they consist of, e.g., the dynamics of the membranes ϕ and the output type of the units. Both, (4.1) and (4.2) above use first order low-pass filters and graded output by means of nonlinearities f. The main difference concerns their connectivity patterns. In pool-models all cells in one pool can potentially reach all cells in the same or another pool – matrix-vector operations are most convenient to implement this kind of model, see section 4.3 below. Neural fields on the other hand reveal topographic neighbourhood structures - Felix provides constructs for the implementation of this kind of "integro-differential equation", too, see section 4.4.

Delays further play an important role in many neural models. They are supported in Felix by a container class that stores model trajectories over time and a number of fundamental routines to access delyed variables in simulations. There are in particular delayed convolution functions, that are needed if lateral propagation speeds in a field model are finite. Details can be found in section 4.5.

Noise is omnipresent in neural systems and in many other physical systems, too. In (4.1) and (4.2) noise inputs into the system is, for instance, represented by the processes η. These are commonly assumed independent and identially distributed Gaussian white noise with mean 0; σ sets the standard deviation. Other choices are Poisson processes of some rate which would reflect the spiking nature of inputs to neurons. Felix has a built-in pseudo-random number generator described in section 4.6.

Felix also has libraries with some numerical and image processing routines. Because, these are not well developed, they will not be described in this document.

4.2 Some Low-level Definitions

If not defined already in system headers, the Felix headers define the following macros

```
# define TRUE 1
# define FALSE 0

# define MIN(a, b) ((a) > (b)? (b) : (a))
# define MAX(a, b) ((a) > (b)? (a) : (b))
```

4.3 Matrix and Vector Operations

The Matrix/Vector functionality is a central part of Felix. Two base-types for variables are in general supported. Most Felix functions operate on scalars, vectors, or matrices of those.

BaseType : floating point values (for historical reasons these are C-type "float"; I don't want to go into the mess if changing to "double").

bBaseType: binary (0/1) values. One bit stored per memory-byte (unsigned char).

BaseType is used for all kinds of continuous cell variables, whereas bBaseType is useful for the representation fo binary vectors of "spikes".

(The bitBaseType available in early versions of Felix is obsolete and shouldn't be used. It used a packed binary format; one bit stored per memory-bit.)

Vector and Matrix-types are derived from the base-types

```
typedef BaseType * Vector;
typedef bBaseType * bVector;

typedef BaseType * Matrix;
typedef bBaseType * bMatrix;
```

Note: Matrices are internally stored as linearized arrays of rows in memory (ie., not as vectors of pointers to rows or columns).

4.3.1 Operations on Scalar Variables

```
BaseType leaky_integrate( float tau, BaseType v, BaseType expr )
```

This macro implements a simple Euler-Scheme for simulating leaky-integrator membranes: $\tau\frac{dv}{dt} = -v + \text{expr}$. Integration stepsize is set with SET_STEPSIZE(dx) and should be chosen such that dx/tau is small compared to 1. The variable "step_size" can be used explicitly in code if required. In conjunction with the later explained fire_reset()-function, "leaky-integrate and fire neurons" are straightforward to implement (see the example program inf.c).

Several basic nonlinear functions are available as rate-functions or for other purposes. They all take a single float as argument and return a single floating point value.

triangle(x) : $f(x) = 1 - |x|$ if $|x| < 1$ and 0 if $|x| >= 1$

rectangle(x) : $f(x) = 1$ if $|x| <= .5$ and 0 if $|x| > .5$

gaussian(x) : $f(x) = exp(-4 * ln(2) * x * x)$ (The factor 4*ln(2) ensures f(.5) = .5)

fermi(x) : $f(x) = 1/(1 + exp(-4 * x))$ (The factor 4 ensures df/dx(0) = 1.)

ramp(x) : $f(x) = 1$ if $x > 1$, 0 if $x < 0$ and x else

lin(x) : $f(x) = x$

tlin(x) : $f(x) = x$ if $x > 0$ and 0 if $x <= 0$

tquad(x) : $f(x) = x * x$ if $x > 0$ and 0 if $x <= 0$

4.3.2 Memory Allocation Routines

Before use, any vector or matrix-variable must be allocated. This is usually done in the top-level function main_init(). It *must* be done there if the variable is accessed for display in the graphical user interface. Use the following template for functions to allocate vectors and matrices:

```
<var_type> var;
var = Get_<var_type>( <dims> );
```

<var_type> stands for “Vector”, “bVector”, “Matrix”, or “bMatrix”. If a Vector-Type is supplied <dims> is the requested length. In case of a matrix <dims> = “rows, columns”.

Allocated memory-space should be set free if no longer need. This is done by macros of the type

```
Free_<var_type>(var);
```

or simply by a call to the system-library function free($< var >$).

Example:

```
Matrix m = Get_Matrix(10, 10);      /* allocate memory for m */
 ......
Free_Matrix( m );                 /* or alternatively: free(m); */
```

4.3.3 Cleaning Vectors and Matrices

All entries of a Vector or Matrix are set to zero by one of the following macros:

```
Clear_<var_type>(<dims>, <var>) .
```

For example:

```
Clear_bMatrix( rows, columns, m );
Clear_Vector( length , v );
```

4.3.4 Access to Elements of a Matrix

```
elem( m, i, j, columns )
```

This is a macro that gets or sets the element $m[i][j]$ of the Matrix or bMatrix m. 'columns' is the number of columns of m. (If i is set to 0 the macro can be used for VectorTypes, too.)

For example:

```
elem( m, 5, 6, 10) = 3.14;  /* Set m[5][6] to 3.14 */
x = elem( m, 5, 6, 10);     /* set x to m[5][6]    */
```

If you don't use this macro for matrices you have to keep in mind that matrices are stored serially in memory, i.e, elem(m, 5, 6, 10) would be equivalent to $m[5 * 10 + 6]$, but note that $m[5][6]$ *does not* work!

4.3.5 Raw I/O of Vectors and Matrices to/from files

Raw output in binary format to or from a stream is done with one of the macros:

```
Write_<var_type>( <dims>, <var>, stream );
Read_<var_type>( <dims>, <var>, stream );
```

These macros directly call the system-library functions fread() and fwrite(), thus they return the number of bytes actually read or written. For error-indications see fread() and fwrite().

For ASCII-output use the functions

```
Save_<var_type>( <dims>, <var>, stream );
Load_<var_type>( <dims>, <var>, stream );
```

Vectors and Matrices are stored row by row; b-Types are stored as sequences of zeros and ones. Entries are separated by blanks. On error the functions return -1; otherwise zero;

There are functions mainly for debugging purposes that print out Vectors and Matrices to stdout in the same manner as the Save-family does to files. These are

```
Show_<var_type>( <dims>, <var> );
```

4.3.6 Vector and Matrix Operations

Scalar Multiplication of two vectors of length n.

```
BaseType Skalar(int n, Vector v1, Vector v2)
BaseType bSkalar(int n, Vector v1, bVector v2)
int bbSkalar(int n, bVector v1, bVector v2)
```

Observe that the purely binary operation returns int-type.

Matrix-Vector Multiplication.

```
Vector Mult(int z, int s, Matrix matrix, Vector vector, Vector dest)
Vector bMult(int z, int s, Matrix matrix, bVector vector, Vector dest)
```

`dest = matrix * vector`, where “matrix” has z rows and s columns, “vector” has length s, and “dest” has length z. The functions return “dest”. Observe that the purely binary operations return int-type, thus a vector to integers has to be supplied as ’dest’ .

Maximum, Minimum, and Sum over Elements.

```
BaseType Sum(int, Vector);
int      bSum(int, bVector);

BaseType Max_Elem(int, Vector);
BaseType Min_Elem(int, Vector);
```

Norms and Scaling The following compute Vector Norms. (Induced) Matrix norms are not implemented at the moment, but matrices can be supplied to the functions below as well.

```
BaseType Vector_Norm_1(int n, Vector v);
BaseType Vector_Norm_2(int n, Vector v);
BaseType Vector_Norm_sup(int n, Vector v);

void Norm_Vector_1(int, Vector v, Vector out);
void Norm_Vector_2(int, Vector v, Vector out);
void Norm_Vector_sup(int, Vector v, Vector out);
```

The "Vector_Norm_" functions compute the 1, 2, and ∞- (or max- or sup-)norm of a vector, respectively (ie, the sum of absolute values, square-root of squares, or the largest absolute element). The "Norm_Vector_" functions first compute the norms and then scale the vectors to a norm of 1. They return the result in "out" which can be the same as "v".

The subsequent functions scale vectors and matrices or apply more general functions to each element

```
Vector Scale_Vector(int n, Vector v, BaseType offs, BaseType scale, Vector out);
Vector Vector_Apply(int n, Vector v, BaseType (*func)(BaseType), Vector out);
Vector Vector_Apply_Arg(int n, Vector v,
             BaseType (*func)(BaseType, void *), void *args, Vector out) );

Matrix Scale_Matrix(int z, int s, Matrix m, BaseType offs, BaseType scale, Matrix out);
Matrix Matrix_Apply(int z, int s, Matrix m, BaseType (*func)(BaseType), Matrix out);
Matrix Matrix_Apply_Arg(int z, int z, Matrix m,
             BaseType (*func)(BaseType, void *), void *args, Matrix out);
```

Scale_Vector and Scale_Matrix apply an affine transformation to the elements of the vector "v" or matrix "m". That is, they multiply all values by "scale" and add an offset "offs".

Vector_Apply and Matrix_Apply apply a user defined function "func" to all elements in the array. The user-defined function "func" takes a single float as input and returns a floating point value; the previously defined non-linearities can, e.g., be used. The function is, e.g., useful to compute the outputs of graded response neurons given their potentials and rate-function.

Vector_Apply_Arg and Matrix_Apply_Arg apply a user defined function "func" with more than a single argument to all elements in the supplied array. "func" takes a void pointer to a vector (or struct) of arguments and must return a single floating point value.

Results of the above functions are return in "out" which can be the same as the input array.

Setting / Changing whole Vectors and Matrices

```
void Set_Func_Vector(int n, Vector v,
                     BaseType (*func)(BaseType),
                     int shift, BaseType height, BaseType scale)
```

This function changes the vector “v” according to $v[i] += height * func((i - shift)/scale)$, where “func” is a scalar function. Note that the function is additive. It can be used to genrate shifted versions of vectors with certain profiles as hey appear as input stimuli in some neural networks.

```
void Make_Func_Band_Matrix(int n, Matrix J,
              BaseType (*func)(BaseType),
              BaseType height, BaseType scale )
```

This generates band-matrices with (row-)profiles given by a function “func”. The function is additive. “height” and “scale” set the amplitude and width of the profile (e.g., if func is a Gaussian, scale would be the standard deviation)

```
void Make_Func_Band_Matrix_Cyclic(int n, Matrix J,
                BaseType (*func)(BaseType),
                BaseType height, BaseType scale)
```

As the previous one this function generates band-matrices with (row-)profiles given by a function “func”, but wraps cyclically. The function is additive.

```
void Dilute_Matrix(int z, int s, Matrix m, BaseType p)
```

This function randomly sets entries in the matrix “m” to zero with probability “p”. A Vector can be diluted by setting one of the size arguments to 1 and the other to the true lebgth of the Vector.

4.3.7 “Neural” Operations for Vectors and Matrices

“Sigmoid” output functions.

```
Vector Fv(int n, Vector vector, BaseType (*func)(),
          BaseType factor, BaseType threshold, BaseType width,
          Vector out)
```

Apply the function func() to all “n” elements of “vector”. func() can be one of the scalar functions defined earlier in this section, or a user-defined one. Result are stored in the vector “out” (not neccessarily different from “vector”). “factor” and “width” may be used to scale the variable-values to the nonlinear range of func(); “threshold” sets an offset-value:

```
  out[i] = factor*func(  (vector[i]-threshold)/width  );
```

The function returns “out”. If out equals NULL (or 0) on entry, an output array is allocated internally and returned by the function. The user has to free() the respective space, if it is no longer needed.

"Poisson" Processes

```
bVector ProbFire( int n, Vector v, bVector out)
```

Computes a n-dimensional binary random-vector from v, such that $prob[o[i] = 1] = v[i]$ and $prob[0[i] = 0] = 1 - v[i]$. It is not checked whether v[i] falls into the range [0,1]. The function returns "out". If out equals NULL (or 0) on entry, an output array is allocated internally and returned by the function. The user has to free() the respective space, if it is no longer needed.

Threshold Neurons.

```
bVector Fire(int n, Vector vector, BaseType theta, bVector out)
```

For all n elements of "vector" compare $vector[i]$ with a threshold "theta": set $out[i]$ to 1 if it is larger and to 0 otherwise. The function returns "out". If out equals NULL (or 0) on entry, an output array is allocated internally and returned by the function. The user has to free() the respective space, if it is no longer needed.

Fire-and-Reset Neurons.

```
bVector Fire_Reset(int n, Vecto vector, BaseType theta,
         BaseType reset, bVector out)
```

Same as Fire() but if $out[i]$ is set to 1 then $v[i]$ is reset to the value "reset". This function may be used to implement the "integrate and fire neuron model" (cf, example program inf.c). If out equals NULL (or 0) on entry, an output array is allocated internally and returned by the function. The user has to free() the respective space, if it is no longer needed.

4.4 Field Models, Spatial Convolutions

Field models are two-dimensional, topographically arranged neural networks, which are typically only connected within certain neighbourhoods, see Figure 1.2.

Although Felix supports one-dimensional and two-dimensional fields, only two-dimensional ones are described in this document.

4.4.1 Kernels or Filters

As stated, cells in neural fields are connected only locally. Felix assumes rectangular connectivity regions, which are called Kernels, or Filters, or receptive Fields. The precise name chosen depends on the context and on the scientific community ("kernels" appear in integro-differential equations in Mathematics, "filters" in image processing algorithms in computer science, and "receptive fields" in neural networks – all three concepts are "very closely related" (to speak cautiously).

```
/* two-dimensional Kernels/Filters */

typedef BaseType * Kernel;
typedef BaseType * UniKernel;

typedef bBaseType * bKernel;
typedef bBaseType * UnibKernel;
```

The difference between Kernels and UniKernels is that in some neural fields all units have the same "filters" (think, e.g., of a layer of cells detecting orientation at a fixed orientation), whereas in others each cell has its own "receptive field" (e.g., in a full orientation tuning map). In the first case one would story only a single copy of the kernel (UniKernel/UnibKernel), whereas in the second case a field of kernels is required (Kernel/bKernel)

4.4.2 Correlation and Convolution Functions

Again somewhat depending on scientific community, operations envolving kernels in field equations are written as "convolutions" $\int k(x-x')f(x')dx'$ or "correlations" $\int k(x+x')f(x')dx'$. The main difference is just mirroring the respective kernel (here shift-invariant UniKernels). Felix implements both options. In neural field applications one would (probably) prefer correlations because they measure the similarity (correlation) of the (input) f with the kernel local at location x.

There are a pretty large number of correlation and convolution functions in Felix, which differ in the types of arguments, and how they deal with the boundaries of a field.

They all take a input field "in" of size x×y and a kernel (Uni or Multi) of size kx×ky; they all return a field "out" of size x×y.

If the local operations are correlations the base name of the function is "Correlate", and it is "Convolute" for convolutions.

If the input field is of binary type (e.g., a field of 0/1 spikes) a "b" is added in front of the base name.

If each local convolution/correlation uses the same UniKernel, "Uni" is appended after the base name. Otherwise, a field of kernels is expected, such that each local unit has its own filter / receptive field.

If the convolution / correlation wraps around at the boundaries, ie., the field is actually a two-dimensional torus, "cyclic" is appended to the name of the function.

Here is the full list of possibilities.

```
Matrix Correlate_2d ( Matrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix Correlate_2d_cyclic ( Matrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bCorrelate_2d ( bMatrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bCorrelate_2d_cyclic ( bMatrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )
```

```
Matrix Convolute_2d ( Matrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix Convolute_2d_cyclic ( Matrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bConvolute_2d ( bMatrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bConvolute_2d_cyclic( bMatrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )

Matrix Correlate_2d_Uni ( Matrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix Correlate_2d_Uni_cyclic ( Matrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bCorrelate_2d_Uni ( bMatrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bCorrelate_2d_Uni_cyclic( bMatrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )

Matrix Convolute_2d_Uni ( Matrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix Convolute_2d_Uni_cyclic( Matrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bConvolute_2d_Uni ( bMatrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bConvolute_2d_Uni_cyclic ( bMatrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
```

All these functions return “out”, which must provide space for the results when a function is called.

Note that the cyclic functions are more time-consuming than the non-cyclic ones, and that UniKernels need less memory.

Later in this section another family of functions is introduced that extends the convolution/correlation functions to include lateral propagation delays, see section 4.5.

4.4.3 Orientation Tuning Maps

The following are a few functions that initialise single UniKernels or arrays of them (Kernels). They can be used to implement orientation tuning maps, but are rudimentary.

```
Set_Circ_Func_Uni_Kernel(UniKernel kern, int kx, int ky,
                BaseType (*func)(BaseType),
                BaseType height, BaseType width, BaseType offset)
```

Given a one-dimensional profile function “func” set a 2d-UniKernel “kern” to a circular symmetric profile. Kernel-dimensions are kx and ky. “height, width, and offset” set the amplitude and spatial scale, and an additive offset of the kernel, respectively. The function is additive.

```
void Gabor_Uni_Kernel ( UniKernel kern, int dimx, int dimy,
                        BaseType height, BaseType sigma1, BaseType sigma2,
                        BaseType kw, BaseType phikw, BaseType phisigmakw,
                        BaseType phi0 )
```

This function sets an UniKernel to have a Gabor-type receptive field, i.e., a 2d-sinusoidal wave modulated a by a spatial Gaussian function. "dimx" and "dimy" are the dimensions of the kernel. "height sets its amplitude. "sigma1" and "sigma2" are the standard deviations of the Gaussian along the first and second principal axes. "kw" is the wave-number. phikw is the orientation of the wave vector and phisigmakw the angle between the direction of the wave vector and the first principal axes of the Gaussian (usually believed to be 0 in cortical simple cells, but need not). "phi0" is the spatial phase of the sinusoidal. The function is additive.

```
void Set_Phi_Func_Kernel ( Kernel kern, int x, int y, int kx, int ky,
                           BaseType (*func)(BaseType),
                           Matrix phi,
                           BaseType height, BaseType width, BaseType offset)
```

This function takes a matrix of orientations, "phi" and generates a two-dimensional field of size x×y of two-dimensional kernels "kern" of size kx×ky. Each kernel has an orientation-tuned profile given by the scalar function "func" in a direction corresponding with the phi-value at the respective location in "phi". (Thus, these profiles can be plane waves, but can not in addition be Gaussian modulated as for Gabor wavelets. There is currently no dedicated function to set fields of Gabor wavelets at once.) "Height, width, and offset" have the same meaning as in the previous functions. The function is additive.

4.4.4 Layers and SpikeLayers

In order to make life easier in some applications, two types of fields have been defined with intrinsically stored sizes, Layers and SpikeLayers. These just redefine the more general structures described above, but use intrinsic variables xsize and ysize for their size.

```
#define SPIKE_LAYER ARRAY_CHAR_TYPE     // same as bMatrix
#define LAYER       ARRAY_FLOAT_TYPE    // same as Matrix

# define DEFAULTXSIZE  64
# define DEFAULTYSIZE  64

# define X_SIZE(_x) xsize = _x;
# define Y_SIZE(_y) ysize = _y;

extern int xsize, ysize;
```

Layers redefine Matrix and SpikeLayers bMatrix. Similarly Fields redefine Kernels and UniFields UniKernels. Default dimensions are 64×64, which can be changed using the macros X_SIZE and Y_SIZE above (in the function main_init()). Thereby, explicit size arguments can be often avoided:

```
Get_Layer()          // returns a Matrix of xsize * ysize
Get_SpikeLayer()     // returns a bMatrix of xsize * ysize
Get_Field(z,s)       // returns a field of xsize*ysize of kernels of size z*s
Get_UniField(z,s)    // returns a single kernels of size z*s

Free_Layer(l)
Free_SpikeLayer(l)
Free_Field(l)
Free_UniField(l)

Clear_Layer(l)
Clear_SpikeLayer(l)
Clear_Field(z,s,l)
Clear_UniField(z,s,l)

Fold_Spikes_Uni(inp, kern, kx, ky, out)
    same as:  bCorrelate_2d_Uni(inp, kern, xsize, ysize, kx, ky, out)

Fold_Spikes( in, kern, kx, ky, out)
    same as:  bCorrelate_2d( in, kern, xsize, ysize, kx, ky, out)
```

4.5 Delays

Delaylines are cyclic buffers that can store values of vectors and arrays of variables from previous steps. The user does not need to mess with the intrinsic data-structures of cyclic buffers. A number of low-level access routines are provided as well as routines commonly encountered in dealing with delays in pool- and field-models.

4.5.1 Containers for Delay Variables

The following are types of container variables that can store different Felix types

```
Vector_DL
Matrix_DL
bVector_DL;
bMatrix_DL;
intVector_DL;
intMatrix_DL;
```

Allocating Delay Lines Use one of the following to allocate a delayline of a particular type. n, r, c are the number of elements, rows, columns, and l is the memory-length, ie, the maximum number of simulation steps that are stored.

```
Get_Vector_DL( _n, _l )
Get_Matrix_DL( _r, _c, _l )
```

```
Get_bVector_DL( _n, _l )
Get_bMatrix_DL( _r, _c, _l )

Get_intVector_DL( _n, _l )
Get_intMatrix_DL( _r, _c, _l )
```

Freeing Delaylines. Delaylines should be freed if no longer used by calling one of

```
Free_DL( _d )
Free_Vector_DL( _d )
Free_Matrix_DL( _d )
Free_intVector_DL( _d )
Free_intMatrix_DL( _d )
Free_bVector_DL( _d )
Free_bMatrix_DL( _d )
```

Note: calling just `free(dl);` is *not* enough. You need to use the above macros. It can be just `Free_DL( _d )`, however, to which all the other macros expand.

Resetting Delaylines. The following macros reset a delayline to a well-defined state; they do not clear the data buffers as such.

```
Clear_DL( _d )
Clear_Vector_DL( _d )
Clear_Matrix_DL( _d )
Clear_intVector_DL( _d )
Clear_intMatrix_DL( _d )
Clear_bVector_DL( _d )
Clear_bMatrix_DL( _d )
Clear_bitVector_DL( _d )
Clear_bitMatrix_DL( _d )
```

Setting Delaylines. The initial values of a delayline can be defined by a function "func" of parameters "P". This has to define values for each vector or matrix element and delay in the delayline. The calls below use the function to initialise a delayline.

```
void Set_Vector_DL( size_t n, size_t del, Delayline dl, float *P,
           BaseType (*func)(size_t x, size_t d, float *P) );
void Set_Matrix_DL( size_t rows, size_t cols, size_t del, Delayline dl, float *P,
           BaseType (*func)(size_t x, size_t y, size_t d, float *P) );

void Set_bVector_DL( size_t n, size_t del, Delayline dl,float *P,
           bBaseType (*func)(size_t x, size_t d, float *P) );
void Set_bMatrix_DL( size_t rows, size_t cols, size_t del, Delayline dl,float *P,
           bBaseType (*func)(size_t x, size_t y, size_t d, float *P) );
```

```
void Set_intVector_DL( size_t n, size_t del, Delayline dl,float *P,
           int (*func)(size_t x, size_t d, float *P) );
void Set_intMatrix_DL( size_t rows, size_t cols, size_t del, Delayline dl,float *P,
           int (*func)(size_t x, size_t y, size_t d, float *P) );
```

4.5.2 Accessing Containers

The following macros select delayed data-containers in a delayline “dl”. It might be necessary to cast types in an application

current(dl): returns a pointer to the container for the current time-slice

last(dl): returns a pointer to the container for the previous time-slice

n_last(dl, n): returns a pointer to the container for the time-slice from “n” slices ago (it is not checked whether n is in proper bounds, ie $<$ memory length.

oldest(dl): returns a pointer to the container for the oldest time-slice (according to the memory length of the delay line

next(dl): returns a pointer to the container for the next time-slice

```
Step_DL(_d)
```

The macro `Step_DL` advances a delay line by one step (time-slice). It must be invoked after the updating of delayline data in the top-level step()-routine. It is assumed that step() stores newly computed data in next(dl) (say, $x(t+h)$ for disretised differential equations or $x(t+1)$ for iterative maps). The routine increments the DL’s current indexes and pointers; i.e recently computed data in “next” become “current”.

4.5.3 Arbitrary Delays for Pools

Communication between two units in a network might take a certain time. In that case the connection is not only characterised by a number (synaptic strength), but in addition by a delay value. The subsequent two functions take delayed float or binary data “in” and multiply them by a coupling matrix “J”, such that each individual connection has a delay as specified by the matrix “delays” (in simulation steps). The results are stored in the Matrix “out”.

```
void  Mult_delayed_DL( int n,
                       Matrix J, int *delays,
                       Vector_DL in, Vector out);
void bMult_delayed_DL( int n,
                       Matrix J, int *delays,
                       bVector_DL in, Vector out);
```

Note: Delays are not checked for falling into range boundaries.

4.5.4 Convolution Functions with Distance-dependent Delays

In two-dimensional fields with local connectivities delays can be distance dependent according to some “axonal” propagation speed and possibly a fixed “synaptic transmittion” delay, too. The following functions generalise the convolution/correlation functions from section 4.4.2 to this case. Naming conventions are the same as there, but _delayed is appended to the function names in the case of finite lateral propagation. The input, of course, now must be a delay line of activities. Arguments “d” and “v” in the functions below are a fixed delay offset (synaptic delay) and the (axonal) propagation speed (in units / time step), respectively.

```
Matrix Convolute_2d_Uni_delayed( Matrix_DL in, UniKernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);
Matrix Convolute_2d_Uni_cyclic_delayed( Matrix_DL in, UniKernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);

Matrix bConvolute_2d_Uni_delayed( bMatrix_DL in, UniKernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);
Matrix bConvolute_2d_Uni_cyclic_delayed( bMatrix_DL in, UniKernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);

Matrix Convolute_2d_delayed( Matrix_DL in, Kernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);
Matrix Convolute_2d_cyclic_delayed( Matrix_DL in, Kernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);

Matrix bConvolute_2d_delayed( bMatrix_DL in, Kernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);
Matrix bConvolute_2d_cyclic_delayed( bMatrix_DL in, Kernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);

Matrix Correlate_2d_Uni_delayed( Matrix_DL in, UniKernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);
Matrix Correlate_2d_Uni_cyclic_delayed( Matrix_DL in, UniKernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);

Matrix bCorrelate_2d_Uni_delayed( bMatrix_DL in, UniKernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);
Matrix bCorrelate_2d_Uni_cyclic_delayed( bMatrix_DL in, UniKernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);

Matrix Correlate_2d_delayed( Matrix_DL in, Kernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);
Matrix Correlate_2d_cyclic_delayed( Matrix_DL in, Kernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);

Matrix bCorrelate_2d_delayed( bMatrix_DL in, Kernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);
Matrix bCorrelate_2d_cyclic_delayed( bMatrix_DL in, Kernel kern, int x, int y,
                            int kx, int ky, float d, float v, Matrix out);
```

Note: It is not checked whether delays fall into proper bounds (must be smaller than the length of the delaylines). The possible axonal speed “v” and fixed additive delay “d” a thereby constrained.

Note further that theses functions are less efficient than their non-delayed counterparts. Cyclic boundaries cause an extra slow-down.

4.6 Random Numbers

Felix has an internal random number generator

Based on 4-state Mersenne twister ? x-check gsl ....

The Felix-intrinsic random number generator should be threadsave if OpenMP or MPI, or mixes thereof are used. However, because the parallel Felix extensions are quite recent, I haven't checked that intensively. (The binomial random number generator is known *not* to be threadsafe for $n >= 25$ and $n * p > 1$.)

Note also, that the initialisation in case of MPI/OpenMP parallel code is very simple. In order to have each thread generate a different sequence of random numbers, all threads contributing to a task are enumerated and the respective thread-numbers are just added to the seed provided to the randomize() function. This can lead to correlations in the numbers generated in different threads. I have no experience yet, how serious the effect can be. Send reports if you run into trouble caused by this overly simple procedure. (I'd then try using /dev/random which, however, is not very portable and has other disadvantageous).

`void randomize( int seed )` initialises the Felix intrinsic random number generator with "seed".

`long rand_long( void )` returns pseudo-random long integers in the range from 0 to $2^{32} - 1 = 4294967295$.

`float equal_noise( void )` returns equally distributed random numbers in the range $[0, 1.0[$.

`unsigned bool_noise( float p )` returns one with probability p and zero with probability $1-p$.

`float gauss_noise( void )` returns gaussian distributed random numbers with mean zero and standard-deviation 1.

`float lorentz_noise( void )` returns lorentz- (or cauchy-)distributed random numbers with mean 0 and standard-deviation 1.

`float binomial_noise( float p, int n )` returns binomially distributed random numbers $B(k; p, n)$ (as float values). [This generator is not threadsafe for $n >= 25$ and $np > 1$. It is mainly intended for implementations of synaptic failure, where n seems to be seldomly above 15 for cortical neuron types.]

Whereas, the previous funtions are all built on the same Felix-intrinsic random number generator, the follwing function (from Press et al) uses its own mechanism to generate random bits.

`unsigned int irbit( unsigned int * iseed )` generates a sequence of random bits, i.e., zeros and ones with equal probability. Iseed is some seed value. The sequences are not "very" random.

4.7 Sparse Vectors and Matrices

4.7.1 Sparse Vectors, semi-sparse Matrices

NOTE: Functions in this section might be subject to later changes as practicality considerations will indicate ....

Code for "sparse" vectors and matrices is currently being developed. Those appear to be useful in very large simulations where cells are only connected with a fraction of other cells. There is support for sparse floating point, binary (char), and integer vectors and matrices. The definitions for the floating point types are:

```
typedef struct
{
  int n,     // actual valid entries
      nmax;  // max entris befor reallocation
  int *i;    // indexes
  float *v;  // values
} sVector_t;

typedef sVector_t *sVector;


typedef struct
{
  int m;      // number of columns
  sVector *w; // array of column vectors
} sMatrix_t;

typedef sMatrix_t *sMatrix;
```

Binary and integer types have an additional 'b' or 'i' in their names, sbVector, siMatrix. These structures are actually "semi"-sparse only. sVectors are sparse, but sMatrices are sparse only in their rows; the array of columns is complete and not sparse. Each such sVector contains the sparse row-entries of that column. This reflects the fact that each neuron in a network projects to at least some other neurons. Similarly, each spike is distributed to at least *some* other cells.

4.7.2 Allocating, Loading, and Saving Sparse Arrays

The following functions corrspond with those for the standard Vector/Matrix types. Not all of these functions are fully implemented at the moment, in especially, none of the FILE I/O functions would work. The latter just print an error message at run-time, when called.

```
sVector Get_sVector( int size )
void Free_sVector( sVector v )
void Clear_sVector( sVector v )
void Empty_sVector( sVector v )
void Show_sVector( sVector v )
void Add_sVector_Entry( sVector, int i, float val )
float sVector_Elem( sVector v, int i)  // returns value v[i] or zero

void Write_sVector( sVector v, FILE*f )
void Read_sVector( sVector v, FILE*f )
void Save_sVector( sVector v, FILE*f )
void Load_sVector( sVector v, FILE*f )


sMatrix Get_sMatrix( int columns, int rows) // order matters
void Free_sMatrix( sMatrix w)
void Clear_sMatrix( sMatrix w)
void Empty_sMatrix( sMatrix w)
void Show_sMatrix( sMatrix w)
void Add_sMatrix_Entry( sMatrix w, int r, int c, float val )
float sMatrix_Elem( sMatrix w, int r, int c )

void Write_sMatrix( sMatrix w, FILE*f )
void Read_sMatrix( sMatrix w, FILE*f )
void Save_sMatrix( sMatrix w, FILE*f )
void Load_sMatrix( sMatrix w, FILE*f )
```

The same functions exist for binary and integer data types with an additional 'b' or 'i' in the names. Most of the function names should be self-explanatory. The difference between `Empty_sVector()` and `Clear_sVector()` ist that the first function just sets the number of active entries in the sVector to zero, whereas the 2cd function sets all active synapse to 0. The same holds for the sMatrix-equivalents.

`Add_sVector_Entry( sVector v, int i, float f)` adds an element with value "f" to an sVector at positions i. `Add_sMatrix_Entry( sMatrix m, int i, int j, float f)` does the same for position (i,j) of an sMatrix "m". If an sVector or sMatrix has to be increased in size, this should happen automatically. The floating point functions are additive – if the entry exists already, the new value is added to the old; for integers and binary data the old value is overwritten.

4.7.3 Sparse Matrix Vector Multiplications

The following are functions that multiply a sparse sMatrix with various other structures like Vectors, bVectors, sVectors, or integer arrays that just contain indexes of units supposed to be momentarily active.

```
Vector sMult, ( sMatrix w, Vector v, Vector out) );
Vector ssMult, ( sMatrix w, sVector v, Vector out ) );
Vector sbMult, ( sMatrix w, bVector v, Vector out ) );
Vector siMult, ( sMatrix w, int n, int *idx, Vector out ) );

Vector sMult_t, ( sMatrix w, Vector in, Vector out) );
Vector sbMult_t, ( sMatrix w, bVector in, Vector out) );
Vector sMult_t_delayed( sMatrix w, siMatrix d, Vector_DL in, Vector out )
Vector sbMult_t_delayed( sMatrix w, siMatrix d, bVector_DL in, Vector out )
```

The "xMult_t()"-functions do transposed multiplication, i.e., multiplication from the left; indexes in a column of a matrix are then interpreted as indexes of units where the respective cells *receive* input from. The xMult-functions in contrast assume that the columns contain *outgoing* synapses of a cell. It should (better) not be assume that any dimensions or arguments are checked. The extra argument in the delayed functions is a sparse matrix of integer valued delays of the same size as the weight matrix. It indicates which entries in the delay line "in" are relevant for a specific synapse.

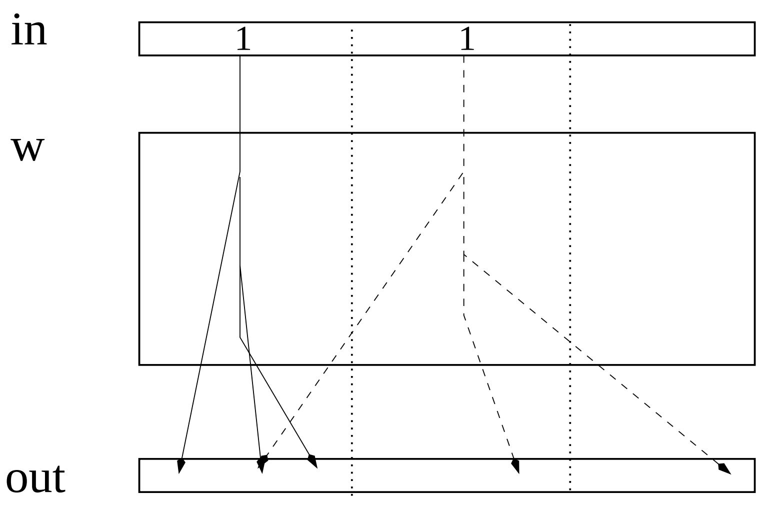


Figure 4.1: Scheme of multiplication of a sparse matrix and a binary Vector.

Figure 4.1 depicts sparse multiplication of a sparse matrix and a binary Vector. An outer loop would run over the input vector. Spikes (1's) in the input array can be distributed in a feedforward way through the matrix, which contains all target indexes and weights. The weights are added to the respective entries in the target vector "out". This, however, can cross thread boundaries (indicated by dashed vertical lines), meaning that the same memory locations are potentially updated by different threads. This can not immediately be parallelised using OpenMP.

Transposed multiplication solves this problem as shown in Fig. 4.2. Here the columns in a sparse matrix are interpreted as containing the indexes and weights of "incoming" synapses to units in the target vector "out". The outer loop then can run over the outputs, in which case each OpenMP-process would update a unique range of entries in the vector "out". Reading from the same location in different threads is not an issue. Even if running on several threads the routine can be less efficient as the previous one on a single thread. This is because it cannot make use of sparseness in the input vector as efficient as the forward multiplication.

Figure 4.3 displays how weights and delays interact in delayed sparse multiplication functions. The sparse delay matrix must have the same dimensions and represent the same connections as the weight matrix. Whereas "w" provides the weights of synapses, the delay matrix determines which element in the input delay line has to be selected. OpenMP parallelisation is again easily possible (and implemented internally).

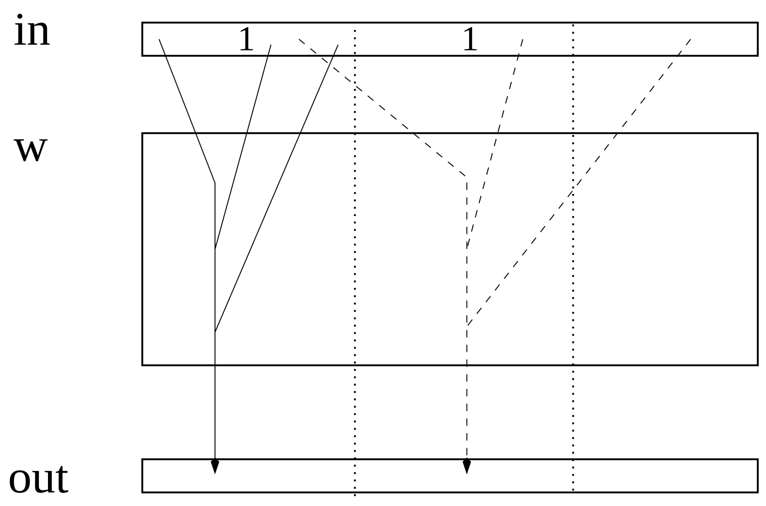


Figure 4.2: Scheme of transposed multiplication of a sparse matrix and a binary Vector.

4.7.4 Orientation Tuning Maps with Distance-dependent Delays

```
sMatrix sCreate_Long_Range_Connectivities(
          int n, Vector in, float scale, float p, float theta );
```

This function takes a feature map “in” of size n and generates a sparse long range connection matrix based on pi-cyclic differences in the features. Synapses are not created if the differences in features are bigger than “theta” (in [0,1[where 1 means ‘identical’). “scale” is an amplitude factor that sets the global scale (applied AFTER “theta”). “p” is an additional probability for creating synapses. Values in the feature map must be in the range [0...PI]. Autapses are not generated;

Note: This function can be used for 1d and 2d-feature maps. 2d-arrays “in” are reinterpreted as one-dimensional arrays of total size “n”. In the 2d-case, however, co-linearity or other “Gestaltprinciples” (beside parallelism) are not taken into account.

```
siMatrix Make_Delays_from_Weight_Matrix( sMatrix w, int xsize, float d0, float v0 );
```

This function takes a weight matrix generated by the previous function and computes a delay matrix from it assuming a 1- or 2-dimensional network topology and distance dependent propagation speeds. If xsize is 0 a one-dimensional topology is assumed, otherwise, “xsize” is the size of the x-dimension in a 2D neural field (the number of columns, ie., total number of units, in the matrix must be a multiple of xsize in that case). d0 is a fixed delay and v0 the propagation delay. Units are in simulation time-steps and lateral units per simulation time respectively. The returned delays will be integers such taht they can be immediately used for indexing elements in a delay line.

The weight and delay matrices returned by the previous two functions can be used in conjunction with the sMult_t_delayed() and sbMult_t_delayed() functions.

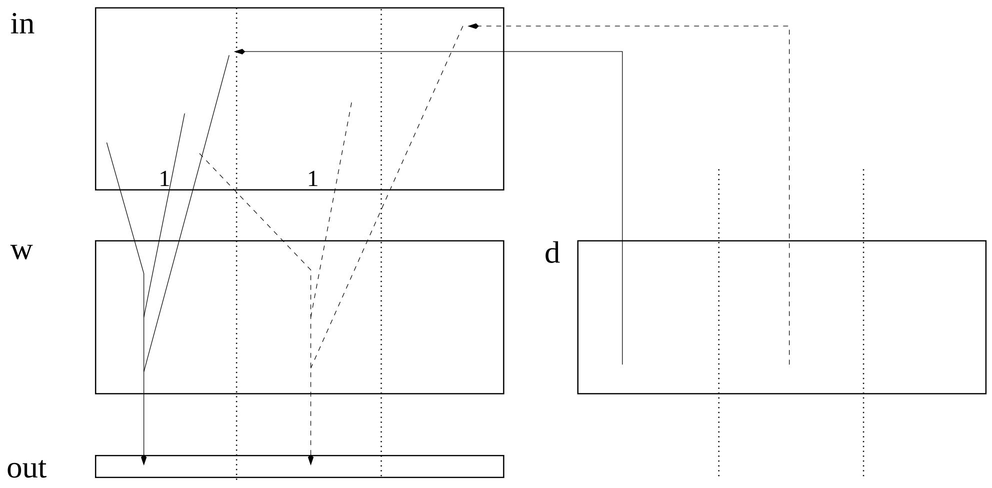


Figure 4.3: Scheme of transposed multiplication of a sparse matrix and a binary Vector with propagation delays.

4.7.5 Displaying Sparse Arrays in the GUI

The graphical user interface of Felix cannot display sparse Vectors and Matrix. You need to convert them befor, using, e.g.,

```
extern Vector Make_Vector_From_sVector( sVector v, int n, Vector out );
extern Matrix Make_Matrix_From_sMatrix( sMatrix m, int r, int  c, Matrix out );
```

"out" must point to memory space of appropriate space when these functions are called. A pointer to "out" is returned. "out" can then be used as usual as an argument to views in the graphical user interface.

4.7.6 Example: Sparse Integrate-and-Fire Network

Here is an example for a leaky-integrate-and-fire network with sparse connectivity. Only ten synpases per neuron/column are allocated from scratch. About a tenth per column are initialised by Gaussian random numbers. Missing synapses are automatically allocated.

Note that the system size is 900, because for small sizes the GUI takes most of the computation time (as long as display windows are open), which is unwanted for proper comparisons. In order to keep display windows at reasonable sizes, we have restricted the maximal sizes in the view declarations.

```
/* Example-program: sinf.c -- integrate and fire network
                              with sparse connectivity matrix */

# include <felix.h>
# include <sparse.h>
# define N   900    /* number of neurons      */
# define tau  10.   /* membrane time constant */

Vector  x;          /* potentials             */
bVector z;          /* vector of spikes       */
Vector  v;          /* auxiliary variable     */

sMatrix spJ;        /* sparse connectivity matrix   <<------------- */
Matrix  J;          /* connections for display */

SliderValue sI     = 100;  /* Common input to units  */
SliderValue sJ0    = 50;   /* Coupling strength      */
SliderValue ssigma = 0;    /* noise level            */

BEGIN_DISPLAY

 SLIDER( "input",     sI, 0, 200)
 SLIDER( "coupling", sJ0, 0, 200)
 SLIDER( "noise", ssigma, 0, 100)

 WINDOW("time courses")

  IMAGE( "x", AR, AC, x, VECTOR,  10, 10, 0.0, 1.0, 4)
  RASTER( "x", NR, AC, x, VECTOR, MIN(100, N), 0, 0.0, 1.0, 1)
  GRAPH( "x", NR, AC, x, VECTOR,  MIN(100, N), 0, 0, 0, -.01, 1.01 )
  RASTER( "out", NR, AC, z, bVECTOR, MIN(100, N), 0, -.01, 1.01, 2)

 WINDOW("couplings")

  IMAGE( "J", AR, AC, J, CONSTANT MATRIX,
                  MIN( 100, N), MIN(100, N), -4./N, 4./N, 2)

END_DISPLAY


NO_OUTPUT

int main_init()
{
  randomize( time(NULL) );
  SET_STEPSIZE( .1 )

  spJ = Get_sMatrix( N, 10 ); // only ten synapses per neuron
                              // are allocated from scratch
  J = Get_Matrix( N, N );
  x = Get_Vector( N );
  z = Get_bVector( N );
  v = Get_Vector( N );
}
```

```
int init()
{
  int i,j;

  Clear_bVector(N,z);
  Clear_Vector(N,v);

  for (i=0; i<N; i++)
    x[i] = equal_noise();

  Empty_sMatrix(spJ);                                        //  <<-----
  for (i=0; i<N; i++)                                        //  <<-----
    for (j=0; j<N/10; j++)  // only N/10 trials per column    //  <<-----
         Add_sMatrix_Entry( spJ, i , (int)(N*equal_noise()) , //  <<-----
                    10.0 / N * ( 1. + .4*gauss_noise() ) );   //  <<-----

  Make_Matrix_From_sMatrix( spJ, N, N, J ); // make a Matrix for the GUI
}

int step()
{
  int i;

  for (i=0;i<N;i++)
    leaky_integrate ( tau, x[i],
                   0.01*( sI + sJ0*v[i] + ssigma*gauss_noise() ) );

  Fire_Reset( N, x, 1.0, 0.0, z );  // firing and reset

  Clear_Vector( N, v );  // need to clear explicitly               // <<-----
  sbMult( spJ, z, v );   // sparse Matrix times non-sparse bVector // <<-----

}
```

Chapter 5

File I/O

The very basics of the file output functionality of Felix have been described in the quick-start chapter 2. We now look a little deeper into the possibilities.

Felix was used over the years mainly to either study autonomous dynamical systems and neural networks, or systems where stimuli could be computed as part of the simulation (e.g., simple bars and graitings). So far, there has never been much need for advanced file-input features and, therefore, Felix provides only some support for *output* of data to files. However, you can always use the standard C methods to load and store data from files (FILE objects, raw and formatted I/O, etc).

Even the file-output properties that are supported are not fully developed. Some facilities, which I imagined would be nice to have years ago, heve actually never been implemented, others never completed. What I describe below are features that I use often or have at least used occasionally.

5.1 Interface for File Output

The philosphy of the file-output interface is similar to that of the graphical display: One has to define a top-level function "MakeOutFiles()", which contains specifications of "OUTFILEs" (analog to "WINDOWs"), which in turn can comprise a variable number of "SAVE_VARIABLEs" (analog to "views on data" or graphics objects in the GUI).

The top-level MakeOutFiles()-routine can be either explicitely defind or constructed by using the macros

```
#define BEGIN_OUTPUT  void MakeOutFiles(){
#define END_OUTPUT    }
#define NO_OUTPUT     void MakeOutFiles(){}
```

Note that NO_OUTPUT expands into an empty function body. In that case no output will be written to external files through the interface mechanisms (but possibly through raw I/O, see section 5.3).

If output files are declared, a button will appear in the graphical user interface, see, Figure 2.4, that actually switches the output on or off during a simulation. The button label reflects the current state. If the button is right-clicked some further control elements appear, which show the files defined, which variables they contain, and some elements that allow to change several file setting interactively. Be aware that not all of the functionality is fully implemented.

Beside using the GUI-Save-button, it is also possible to switch file I/O on or off from the source code by using the macros

```
SAVE_ON
SAVE_OFF
```

Another macro that often is useful influences the format of ASCII output. The macro

```
SET_ITEM_SEPARATOR( sep )
```

takes a string and inserts it between subsequent entries in the output. The macro should be placed right after the head of the the MakeOutFiles()-function. Default for the item separator is a single blank (`" "`), but this can cause problems with very long linelengths in files that have to be read from another program. Some tools for postprocessing (e.g., gnuplot) also expect only a single entry per line (by default in some modes), in which case the item separator can be set to newline (`"\n"`).

5.1.1 Output Files

Inside the function MakeOutFiles() one or more output files have to be defined using the macro

```
OUTFILE(name)
```

where “name” is the name of the file. If the file does not exist and data is written, it will be created, otherwise the old file will be overwritten.

The macro OUTFILE returns a file handle of type `int`. It is not often necessary to save the handle, but some of the later functions make use of it.

Output files are declared in serial order (as WINDOWs in the GUI). Instead of the file handle one can also use the macro `THISFILE`, which expands to the currently active file (ie the most recently declared one).

The file handles are only necessary if an application needs to set file-properties explicitly. The macros

```
FILE_ACTIVE( fileno )
FILE_INACTIVE( fileno )
```

anywhere in the code can, for instance, switch file-output to a particular file on or off. (This, however is further controled by the global SAVE_ON/SAVE_OFF switch. As long as that “master” switch is off, nothing will be saved.)

Other file properties are file format (raw (default) or ASCII)) and the behaviour in case the file is switched on and off more than once in a simulation (data can be overwritten or appended). These flags are set using

```
SET_SAVE_FILE_FLAG(fileno, flag, val )
```

where fileno is the file-handle (or "THISFILE"), flag is "ASCII" for declaration of the output mode and "APPEND" for the reset mode. Possible values for "val" in both cases are ON and OFF, i.e. `SET_SAVE_FILE_FLAG(THISFILE,ASCII,ON)` would switch ASCII output on for thelast recently decalred file in the MakeOutput()-function. (Note that it does not make sense to switch between both modes during one simulation. The files would then at least be relatively difficult to read; depending on the platform/C-implementation results can even be undefined).

The file flags should be set right after the declaration of an output file, ie, before any output variables.

If ASCII mode is on, an empty line will be saved after each vector or row of a matrix, and an extra newline after each complete matrix. The current step will also be saved on an individual line starting with the double-cross # befor all other data in that step. No such extras are saved in raw mode, just pure binary data.

5.1.2 Output Variables

Each output file can contain a number of output variables declared by the macro

```
SAVE_VARIABLE(name, var, type, dim_x, dim_y, flags, when, which)
```

Meaning of the argments is very similar to the various graphical views on data (see, section 3.3.2).

"name" is a string for the name the entry appears under in the graphical user interface.

"var", "type", "dim_x", and "dim_y" are the variable to store, its type, and dimensions. The types and specification of dimensions are the same as for graphics objects in display windows (MATRIX, VECTOR, etc.), see section 3.3.4. POINTER types are possible.

"flags" are output variable-specific flags that are mainly used to specify which data entries are stored when. Default is that each value is stored in each step (as long as the gobal save switch and the respective file-switch are ON)

"when" and "which" are further used to declare spatial and temporal selections of data to store in detail. This is useful in large simulations where output files can easily become very large. The options for sub-selections are explained in the subsequent two sections.

5.1.3 Temporal Selections

By default (and only if the save-button is activated) data is saved after a call to the top-level init()-function (to save "initial values") and after every simulation step. This can be modified individually for each SAVE_VARIABLE using the "flags and when" arguments in their declaration.

A CONSTANT variable that doesn't change during a simulation can be declared by an ONINIT flag. Such a variable is then only saved after calls to init(), because there is where it would naturally be initialised. Possible flags are:

```
ONINIT

SKIP
RANGE
SELECT
```

The last three flags correspond with three functions as arguments to the "when"-argument of the SAVE_VARIABLE declaration:

TSkip(skip) : Only every "skip" step is stored

TRange(start, stop, skip) : Data is stored at regular intervals starting at time step "start", storing every "skip" steps, up to a maximum step of "stop"

TSelect(n, vals) : "vals" is an integer array of size "n" that defines points in time when the data has to be saved.

A few examples are shown in subsection 5.1.6.

5.1.4 Spatial Selections

As in the temporal domain, selections can also be made spatially, more precisely, in one- or two-dimenional arrays. By default, *all* entries in an array-variable (MATRIX, VECTOR, etc.) are stored, if the temporal selection permits it. Alternative options are GRID, IRR_GRID, or POINTS, which refer to regular grids, irregular grids, and sets of individual points/coordinates, respectively.

As for the temporal selections the spatial selection (if it is not ALL) has to be notified in the flag-argument of a SAVE_VARIABLE (see above) using one of

```
GRID
IRR_GRID
POINTS
```

The precise selection has then to be specified as the final "which"-argument of a SAVE_VARIABLE declaration using one of the corresponding functions

Grid(start, stop, skip, start2, stop2, skip2) : This can be used for regular subgrids. "start, stop, and skip" are the first and maximal index of stored elements in the first dimension (x) and "skip" is the regular interval between indexes. The same meaning applies to "start2, stop2, and skip2" in the second dimension (y). For one-dimensional arrays start2, stop2, and skip2 should be zero.

Irregular(nx, values_x, ny, values_y) : This defines an irregular grid, where the integer array "values_x" contains "nx" coordinates in the first dimension and likewise for "ny, values_y". Data is saved for matrix entries at all pairs of x and y values. For one-dimensional arrays the ny and y-values should be zero.

Points(n, values_x, values_y) : This is the most general option because it allows for arbitrary coordinates in the index (integer) arrays "values_x, values_y" of size "n". Data values at the respective n points are saved. For one-dimensional arrays "values_y" should be zero.

A few examples are shown in subsection 5.1.6. Index boundaries are not checked. It is the programmers responsibility to make sure indexes do not exceed array-dimensions. Order for two-dimensional Grid() and Irregular() grid data is left-right (x first), then top-bottom (y).

5.1.5 The Timer

The timer (or Stop Watch) is a further facility to control when storage of data starts and ends. It can, for instance, be used if you want to skip a number of steps at the beginning of a simulation befor saving data because they are transients. Another reason is to set a global skip-interval on top of the temporal selections for the individually saved variables. That can be desirable if the amount of data generated is very big, but storing less steps would already be sufficient. To setup the timer use

```
SET_SAVE_TIMER( start, end, skip )

TIMER_ON
TIMER_OFF
```

`SET_SAVE_TIMER` only sets the parameters of the timer, i.e., the first and maximal step it tries to save anything, "start" and "end", and the interval (in simulation steps) at which data is stored, "skip". If it is to be used, the timer has to be enabled explicitly, either from the GUI by right-clicking on the Save-button and selecting the appropropriate tick-box or by calling TIMER_ON from the source code. It furthermore only generates output if the global save switch is on in addition (the master fuse for your valuable hard disk space).

Observe that the GUI also allows to set the parameters of the timer ("Stop Watch") by hand; they do not need to be set in the code.

5.1.6 Examples

```
int nx=3, ny=2;
int xsel[3]={1,2,5};
int ysel[2]={3,4};

BEGIN_OUTPUT

  SET_ITEM_SEPARATOR( "\n" )

  // 1. example

  OUTFILE("patterns")
    SAVE_VARIABLE( "pats", pats, ARRAY_INT_TYPE, Nones, P, ONINIT, 0, 0)
  // 2. example

  OUTFILE("Quality")
    SAVE_VARIABLE( "qual", &Q, FLOAT_TYPE, 0, 0, 0, 0, 0 )

  // 3. example

  OUTFILE("file42")
    SET_SAVE_FILE_FLAG( THISFILE, ASCII, ON)
    SAVE_VARIABLE( "z", z, bVECTOR, N, 0, 0, 0, 0 )
    SAVE_VARIABLE( "phi1", pot1, MATRIX, xsize, ysize, 0,0,0 )

  // 4. example

  OUTFILE("phi2")
    SAVE_VARIABLE( "phi2", pot2, MATRIX, xsize, ysize,  SKIP | GRID ,
                   TSkip(2), Grid(38, xsize, 100, 32, ysize, 100) )

  // 5. example

  OUTFILE("phi3")
    SAVE_VARIABLE( "phi1", pot1, MATRIX, xsize, ysize, IRR_GRID
                   0, Irregular( nx, xsel, ny ysel ) )

END_OUTPUT
```

In the example the item separator is first set to `\n` such that individual entries go to separate lines.

The first example defines an output file "patterns" to which an integer array "pats" of size Nones $\times$ P is stored once after each call to the top-level init() function (selected by the ONINIT flag). There are no further spatial or temporal selections.

The second example stores a single floating point variable "Q" in each step to a file "Quality".

The third example – in contrast to all others – stores data in ASCII format because the respective flag is set. Data goes to a file "file42". Stored per step are a binary vector "z" of size N and a matrix "pot1" of size $xsize \times ysize$ without any further spatial or temporal restriction.

The fourth example stores a matrix "pot2" of size $xsize \times ysize$ to a file "phi2". Only every second time step is stored and the matrix is spatially sub-sampled on a regular grid.

The last example subsamples a matrix on an irregular grid, but there is no temporal selection.

5.2 Input

A graphical user interface for input from files is not available and not planned. Raw file input functionality has to be used instead (see next section).

5.3 Raw I/O

Instead of using the graphical interface for I/O operations, those can be included directly in the application program using the usual C file access options (see textbooks on C-programming).

Chapter 6

Parallel Programming with Felix

NOTE: This chapter is extremely preliminary

The present chapter describes recently developed parallel computing extensions to Felix. They are under development and many of them barely tested. Use at your own risk and don't expect too much!

Felix supports three types or parallelism: SSE-extensions, OpenMP for symmetric multi processors, and the message passing interface (MPI). The underlying concepts of these three technologies will be described in the subsequent chapters 6.2 to 6.4 individually. However, it is possible to combine all three frameworks in a single program. This makes sense in especially on computer clusters where each single node has several processor cores (see section 6.5). Such clusters will likely be the standard in future computer clusters.

Felix programs can be developed to run on serial or parallel target architectures. In general, at least some effort is necessary to parallelise a given serial code. However, it is at least in principle possible to write Felix programs that can be compiled and run on both, parallel and serial computers. Section 6.6 gives advise on how to write Felix programs of this kind.

6.1 History and Future

The very first Felix version was mainly intended to provide a graphical user interface for a parallel computer we had at the University of Ulm/Germany in the early 90th of the previous century (yes, I am almost a hundred years old!). This was a so-called "WaveTracer" comprising 4096 single bit processors running at an amazing 8MHz cycle-frequency. The processors could be arranged to form 1, 2, or 3-dimensional virtual arrays aiming primarily at simulations of wave equations and partial differential equations; the simulation of neural field models was possible as well. The programming made used of an ingenious C-dialect called "Multi-C", which I still believe was a brilliant development: It was C, enriched by a handful of parallel constructs for parallel data-types and data-transfer between nodes. Unfortunately the company WaveTracer died after a while and as it seems none of the other parallel hard- or software developers took over the conceptual ideas the WaveTracer system incorporated.

When single-CPU computers got faster than the WaveTracer, which happended surprisingly quickly, Felix was ported to standard architectures, first Sun-Workstations under Sun-OS and early Solaris versions, later Linux PCs (and even later Cygwin ...).

More recently, computer clusters got cheap enough to become available for academic research. This caused Felix to be (back-)adapted to parallel environments again. The parallel Felix extensions therefore are very new, meaning that they are neither complete, nor very well developed, nor tested to a degree they should. So, be warned! In fact, they are under development and get extended as I find it useful for my research.

Felix supports three types of parallelism which intentionally should be freely combinable in applications. These technologies are abbreviated as SSE, OpenMP, and MPI – the first is a hardware technology for code-vectorisation, that latter two software-standards for the programming of symmetric multi-processor computers (SMP) and computer grids and clusters, respectively. None of them requires that you actually have a special parallel computer. You can install the necessary software on any Linux-box. This would allow you to develop parallel software on a Laptop or workstation, befor going big on a cluster. In fact, even better, every modern Intel or AMD processor supports SSE intrinsically, and the dual-core processor computers that currently start conquering the market have two physical processing units (SMP) per CPU-chip; they can naturally be programmed using OpenMP (or MPI) if full use of the two processor cores has to be made. Imagine that two cores per CPU are just the beginning: Intel has already presented its first 80-core wafer prototype and others will follow; 4 or 8 core CPUs will probably be available commercially in just a very few years.

6.2 SSE, BLAS, ATLAS

SSE is a hardware technology supported by each mordern AMD or Intel CPU. It was originally invented by Intel to speed up graphics and audio applications, ie., computer games, video, and all that kind of applications companies really can make money with.

SSE is indeed something like a co-processor in every single modern Intel or AMD CPU (I am not sure about MACs; but they switch to Intel CPUs as it seems). Each such processor has a main central processing unit which supports a certain instruction set and is most active during the execution of any standard program. Virtually all modern CPUs in addition have a math co-processor which can be used for speeding up computations of various mathematical functions like abs, sin, exp, and so on. Less well known is that since the Intel 386??? family or AMD ??? each processor has a further processing unit independent of the main arithmetic-logical-unit and math-co-processor that is useful for some kinds of parallel computations appearing often in graphics and audio processing. This hardware piece on modern chips is programmed by using the so-called SSE-extensions to the low-level assembler instruction set for that CPU.

The SSE standard basically provides a special register set on the CPU and accompanied assembler instructions which support some kind of math (but not a whole lot) supposed to be useful for graphics and audio applications. These register (by default 8 of them) are (at least on a 32 bit architecture) 128 bit wide, but the 128 bit can be divided into data-chunks of various size, ie., singned and unsigned integers of 8, 16, or 32 bit size, but also floating points of size 4 or 8 bytes (32 or 64 bits). Accordingly, these special units on any modern Intel or AMD CPU (yes, I am probably speaking about your computer) are able to process up to 16 8-bit integers, or 8 16-bit integers, or 4 32-bit floating points, or 2 64-bit floating points (doubles) at once. This supports a kind of “vectorisation”, operations can be done in parallel on several numbers (ie., a “vector”) at once. In principle every software could make use of this vectorisation, and indeed, commercial compilers as well as newer versions of the gnu compilers are potentially able to compile code written in a higher programming language to make efficient use of the SSE extensions. (A full description of the SSE standard can be found in the respective documents available from Intels web-pages.)

Now, the "BLAS" is the so-called "Basic Linear Algebra Subroutines"-package which is available for Linux (MAC and Windows quite surely, too). It is a highly optimised package of linear algebra routines such as scalar, matrix-vector, and matrix-matrix multiplications. Some commercial products like Matab make use of the BLAS, which make their Matrix/Vector routines very efficient.

A standard Linux distribution does not usually have by default an optimised BLAS, because that library needs to be adapted to the precise target architecture. Most default Linux systems just have a default library (compiled for i368) that can be used by all pre-compiled programs on 99.9% of all PC architectures that need the library. However, you can update your BLAS to speed up such programs. Most of the improved BLAS versions do make use of the SSE extensions.

Two BLAS implementations are kind of standard at the moment: ATLAS- and Goto-BLAS.

ATLAS is an "automatically tuned linear algrebra system" that provides a BLAS and some routines on top of that (a subset of "LAPACK", a well-known "Linear Algebra Package" for solving linear equations, finding eigen-vectors, etc.). During compilation of ATLAS-BLAS, out of a large number (sometimes several hundreds and more) of possible implementations for a particular task like matrix-vector multiplication the best performing routines for the target architecture are chosen and put into the library. These top-performing routines can make use of the SSE CPU extensions and therefore the BLAS is mentioned under "parallel" Felix extensions.

GotoBLAS is a second BLAS implementation originally developed by Kazushige Goto. It is available (ie optimised) for a variety of target architectures and generally said to be the fastest BLAS implementation available. It does use hand-optimised (SSE-)assembler code.

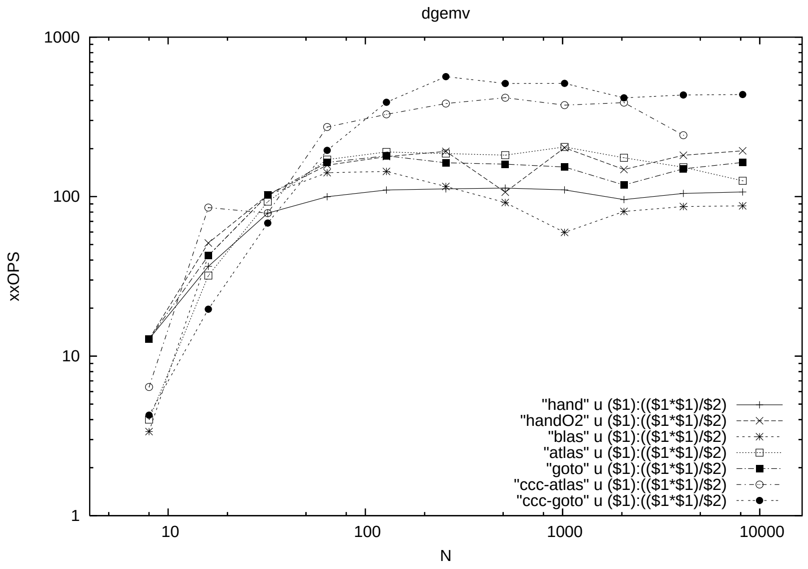


Figure 6.1: Typical performance for floating point *matrix-vector* multiplication on a Centrino laptop and a 4-core AMD opteron machine (CCC) of different raw and BLAS enhanced codes. x-axis indicates matrix/vcteor-size. See text for further explanations.

Figures 6.1 and 6.1 show the performance of matrix-vector and matrix-matrix multiplications for

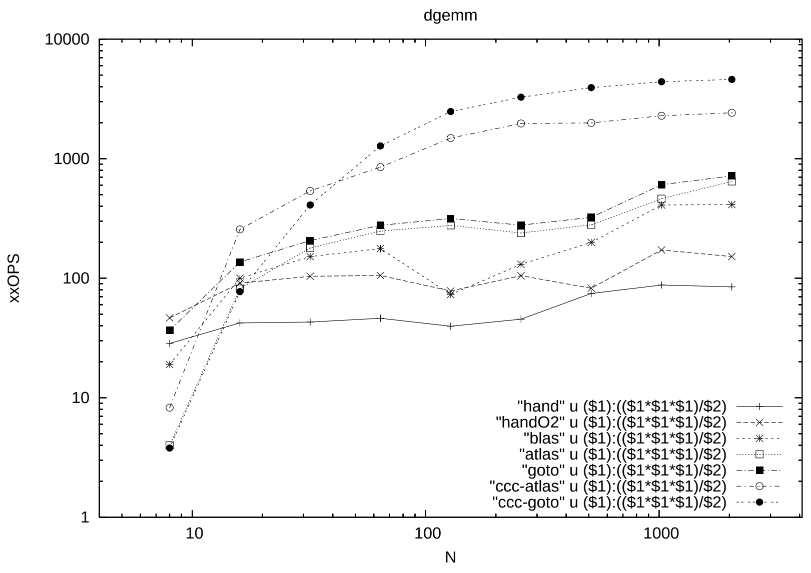


Figure 6.2: Typical performance for floating point *matrix-matrix* multiplication on a Centrino laptop and 4-core AMD opteron node (CCC) of different raw and BLAS enhanced codes. x-axis indicates matrix/vcteor-size.

floating point linear systems under different conditions. "ccc-goto" and "ccc–atlas" have been run on two processor dual core AMD 27?? nodes (4*2GHz; 4*1MB cache???); the other curves are for a single CPU Centrino Laptop (1.73GHz, 2MB cache). "hand" and "handO2 denote naive code (straight for-loops) either compiled without or with O2 optimisation using gcc 4.0.2. "blas" uses the default BLAS library on the Laptop, which performs worse than no optimisation at all in most of the studied range. "atlas" and "goto" indicate ATLAS and Goto-BLAS versions on the respective systems. Observe the quite impressive performance gain for optimised code, and that, of course, the numbers for the Centrino-Laptop and 4-core high-performance compute node are not directly comparable. Interestinly enough for small system size the single-CPU Laptop is faster than the 4-core AMD node.

Note: You don't need any BLAS library if you want to use Felix. It just can make some routines faster. At the moment the numbers of routines that potentially use BLAS-calls is actually more restricted than it could be. However, the floating point scalar-products, and matrix-vector products do use a BLAS library if this has been specified during compile time of the Felix libraries.

In order to let the Felix core use BLAS-routines whereever this is implemented to date it suffices to specify the -DWITH_BLAS flag during compile time. BLAS should not spawn threads (Goto-Blas can do this. It can be avoided using environment variables. See the repsective BLAS documents.)

6.3 OpenMP

Symmetric multi-processor computers (SMP) are systems that comprise a number or central processing units but share a common memory pool. Each processor can access the memory through a fast bus making memory access and data exchange potentially very fast.

Only since relatively recently SMP computers have been developed for the general market at reasonable prices. Meanwhile, however, dual- and quad-processor computers are available at quite low prices and dual-core processors indicate a new trend that even aims at putting two (or more) processing units on the same chip. There are already many dual-core machines available, including Laptops. These all are SMP computers. Linux should support them automatically if you install an SMP-kernel.

OpenMP is an industry standard that supports programming SMP computers. It is not the most general approach for parallel programming (cf., e.g., concepts like Posix threads, PVM, or MPI), but for some kinds of applications it is *very* simple to use and can provide good speed ups. This includes neural network applications.

The most typical example for OpenMP-parallisation is "outer-loop"-parallelisation. It often is possible in numerical code where the same operations have to be performed on a large number of units. This is typically done in a big "outer" loop over the elements. OpenMP provides simple constructs to cut such loops into pieces of roughly the same size and distribute them over the availabe processors. In principle as single additional statement on top of an existing for-loop can be enough to parallelise it, e.g., a statement like

```
 for (i=0; i<N; i++)
   x[i] = func(i);
```

could result in

```
# pragma omp for private(i)
 for (i=0; i<N; i++)
   x[i] =  func(i);
```

This second version is automatically compiled into code distributed over the available processors. The variable i is declared private, because each process will need an independent copy of it. There are other constructs available for more general programming constructs than for-loops. There are also serious constraints that have to be taken into acount when parallelising code – for short, no two processes should ever try to potentially update the same variable at the same time (for that reason i has to be declared private in the pragma-statement; of course the function "func()" also is supposed to not assign values to variables possibly overlapping between processes. If this happens (a so-called "race-condition") the results of the computation are undefined. There are many cases, however, where assignments of variables are constraint to contiguous regions, e.g., in the range of a for-loop. In that case OpenMP-parallelisation is in general save to use. For further details, we have to refer the reader to the OpenMP specification, handbooks, and tutorials available in the Web.

A large number of functions in the Felix core automatically make use of OpenMP parallelisation if this is specified during compile time. It is also possible to use OpenMP macros in the user defined top-level init() and step() functions. It should, however, be avoided to call already parallelised Felix routines in parallelised regions in the top-level functions. Although the results would (probably) be well-defined (up to possible race-conditions), the code would spread an unnecessary large number of threads. Spreading threads for parallel computation always needs some overhead. It is therefore usually not advisable to parallelise very simple loops or to spawn more threads than processors are available.

6.4 MPI

MPI, the “Message Passing Interface”, is an industry standard for communication between processes. These processes can run on the same or different computers, no matter where they are located (physical location only impacts the communication speed). Thus, MPI is useful for computer clusters and grids.

Simple MPI programs make use of a handful of statements only, although the full MPI standard defines over 120 different functions. These most simple commands just set up a logical network, and send and receive messages between nodes. For MPI-details in programming and usage we refer the reader to the many tutorials about MPI programming available on the Web. The following assumes basic knowledge about MPI programming.

Felix provides very simple constructs that don’t do more than exchanging packages of various types of variables between processes.

The general philosophy is to run a number of copies of *the same program* on a number of available nodes (e.g., with `mpirun -np 3 programname` in the usual way to run MPI programs). Each copy has associated with it a number “myrank” that identifies it uniquely. Inside each running process code can therefore be executed conditionally depending on the rank of the process. After each simulation step, variables that are computed inside one process, but required in the next in another process have then to be communicated using MPI.

For that purpose every MPI-parallel Felix programm has to define a top-level routine “fmpi-connections()” that specifies which data has to be communicated. For each variable to be transmitted between two nodes a connection has to be setup using

```
void fmpi_connect( int node1, long var1,
                   int type, int size,
                   int node2, long var2 );
```

or the equivalent macro CONNECT (see example below).

“node1, var1” specify the source variable (typically an array of type CHAR, INT, or FLOAT, but can be a pointer to such an array, too, see below)

“node2, var2” is the target variable (must be an array, no POINTER type)

“size” is the number of elements in the array that have to be transmitted

“type” is the type of the data. The data basetype must be one of CHAR_TYPE, FLOAT_TYPE, or INT_TYPE. Note that Felix Vectors and Matrices are FLOAT_TYPE and bVectors CHAR_TYPE, such that it is admissable to specify the type as, e.g., bVECTOR or MATRIX. The basetype of the target variable must match that of the source. However, the source can in addition be a pointer type (similar as for display variables).

All connections have to be defined in a top-level function fmpi_connections(), e.g., like this:

```
void fmpi_connections()
{
  CONNECT( 0, var1, VECTOR, N, 1, var2 );
  CONNECT( 1, z1, POINTER TO bVECTOR, N, 2, z1 );
}
```

The first statement connects the float vector var1 of size N on processor 0 to var2 on processor 2. The second statement uses a POINTER variable, which gets dereferenced just befor transmission to a bVector of size N which is then transferred from processor 1 to variable z1 on processor 2. It is possible that source and destination are the same variables, but note that they will reside nonetheless on different machines.

Furthermore, if the same code is compiled using serial Felix, the CONNECT macro translates to empty code (but not the function, so use the macro!). Thereby, the code is discarded; nothing needs to be communicated if the program runs on a single processor. This supports writing code that can be compiled on serial *and* parallel machines without changing a singe line. Of course, using this feature needs a careful design of the code in order to have the serial and parallel codes consistent. There is typically at least a one-simulation-step delay introduced, because in the parallel versions communication occurs only after each simulation step, whereas in a serial program updated variables are immediately available.

6.5 Hybrid MPI/OpenMP Code

MPI and OpenMP can be combined in the same program.

The common free MPI versions (MPICH and LAM) are not threadsafe (most commercial implementations are). This means, if you use MPI within OpenMP parallelised regions the results are undefined.

Nonetheless, writing hybrid MPI/OpenMP-programs is possible, if care is taken of calling MPI-constructs only in OpenMP serial parts of the code. In that case only a single thread is doing the MPI-communication, which is safe.

Hybrid parallelism is possible in Felix. For that a number of MPI-processes are spawned that communicate as explained in section 6.4, but each of these processes in turn can spawn their own OpenMP threads. This is useful on SMP clusters with several CPUs per node. Communication between nodes can that way be done using MPI, but on the same node using threads and shared memory. Because communication via shared memory is usually faster than via a network this should result in speed benefits.

The following code is NOT Felix but just a simple C-example that demonstrates the principle.

```
// ompi.c -- simple test program for hybrid MPI/OpenMP paralellism

# include <stdio.h>
# include <omp.h>  // include OpenMP header
# include <mpi.h>  // include MPI header

# define NUMTHREADS 3  //  set number of OpenMP threads here

main( int argc, char *argv[] )
{
  int numtasks, rank;
  MPI_Init( &argc, &argv );
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks );
  MPI_Comm_rank(MPI_COMM_WORLD, &rank );

  omp_set_num_threads(NUMTHREADS);

# pragma omp parallel
{
  printf("MPI rank %d OMP thread %d\n", rank, omp_get_thread_num());
}

  MPI_Finalize();
}
```

The code needs to be compiled with an OpenMP-capable compiler (Intel, gcc 4.2 or higher) and linked against the proper MPI-libs (see also section A for further low-level info). It can then be run using, e.g., `mpirun -np 2 ompi` if "ompi" is the name of the executable. The number of MPI processes in the example would be 2 (specified by "-np 2" in the mpirun call), each of which spawns NUMTHREADS OpenMP threads. Each thread prints its MPI rank and thread number and exits.

Note that instead of setting the number of OpenMP threads explicitly one could also use the environment variable OMP_NUM_THREADS. This is quite usual and avoids having to recompile the code for different thread numbers. However, even if OMP_NUM_THREADS is set in your .bashrc, it is not necessarily exported to all target machines on all systems.

6.6 Parallelising Serial Felix Code

Serial code is compiled using the standard "Felix"-script, which links against libf (core routines) and libxf (XView extensions). For parallel code use the "pFelix"-script. This links against libpf.

Although libf and libxf are for serial code they can possibly make use of BLAS or OpenMP depending on how they have been compiled.

The parallel Felix lib "libpf" must be used for MPI and hybrid MPI/OpenMP.

6.6.1 OpenMP and *pflx*

To make life easier a couple of macros have been declared for writing parallelised code. If you use them you can even write programs that can be compiled and run with and without OpenMP.

```
# ifdef WITH_OMP
# define OMP_THREADS(_n) omp_set_num_threads(_n);
# define OMP_FOR(_x) ... // this shouldn't occur because preFelix removes OMP_FORs
# define OMP_ONLY(_x) _x
# else
# define OMP_THREADS(_n)
# define OMP_FOR(_x) for(_x)
# define OMP_ONLY(_x)
# endif
```

Observe that depending on whether the flag WITH_OMP is active during compile time (usually set in the Makefile) the macros expand into different code. If the Felix-script is used for compilation WITH_OMP will (usually) not be defined, but for the pFelix script it will.

Note that these settings only apply to your source code. Whether OpenMP is used in "libf", the Felix core library, depends on the value of OpenMP at compile time of the libraries, ie. in the Makefile in the Felix source directory. In the standard installation, libf would not contain OpenMP parallelised code.

There are several problems with the OMP_FOR macro: Actually this must have the form

```
OMP_FOR( <var> = <code> )
  < single statement or code-block enclosed by {}>
```

It should expand into

```
#pragma omp parallel for default(shared) private( <var> )
for( <var> = <code> )
  <single statement or code-block enclosed by {}>
```

The code segments not explicitely specified, of course, must translate into valid C-code.

The first problem now is that the `#pragma` phrase cannot be inserted by the preprocessor (at least I don't kow how to do it with macros). Instead a very simple preprocessor call "pflx" is used. This does nothing but searching a file for the string OMP_FOR and replacing the string in the sense above. "pflx" is called, when the Felix- or pFelix-scripts are executed. It generates a temporary file, which is then compiled into an executable.

The second problem with OMP_FOR is that the user has to make sure that the source-code does not contain assignments to memory locations which are potentially executed simultaneously in different threads. The values of such variables are undefined, but there will be no explicit warning or error message. Such variables in general need to be declared "private" in the reprective enclosing OpenMP-pragma or protected by other means (see OpenMP handbooks and tutorials). The only variable that is explicitly declared private if the OMP_FOR macro is used, is the run-index of the for-loop. This suffices in many situations I have experienced over the years. However, if you have code where some threads would potentially write/change the same shared memory locations, you can not use OMP_FOR, but have to use the original OpenMP pragmas.

Example: The following code is buggy

```
int i, j;
Matrix x;  // size nx * ny; allocated elsewhere
...
OMP_FOR(i=0; i<nx; i++)
  for (j=0; j<ny; j++)
    x[i*nx+j] =  ... something ... ;
```

The mistake is that j is a shared variable (by default). If several threads execute the outer for-loop, they all use the same copy of j (in shared memory), which they update asynchronously. This situation occurs often in simulations of two-dimensional field model. A simple cure is

```
int i;
Matrix x;  // size nx * ny; allocated elsewhere
...
OMP_FOR(i=0; i<nx; i++)
{
  int j;
  for (j=0; j<ny; j++)
    x[i*ny+j] =  ... something ... ;
}
```

Here the variable j is local to each thread and can only be changed by the respective thread. x is also a shared variable which gets values assigned, but note that the entries in that matrix are disjoint between threads, because different threads operate on different slices of the matrix.

6.6.2 MPI

A number of macros support writing MP-code.

```
# ifdef WITH_MPI
# define RANK(_x) if(myrank==(_x))
# define COND(_x) if(_x)
# define MPI_ONLY(_x) _x
# else
# define RANK(_x) // if(myrank==(_x))
# define COND(_x) // if(_x)
# define CONNECT(_x1,_x2,_x3,_x4,_x5,_x6)
# define MPI_ONLY(_x)
# endif

extern int myrank;
```

Observe that these macros expand to empty code when compiled serially (ie, if the compiler flag WITH_MPI is not set (usually in the Makefile, see **??**))

RANK and COND support conditional execution of code in conjunction with the global variable “myrank” which holds the unique MPI-rank of each process.

MPI_ONLY() can be used to enclose code that has to be executed only in an MPI environment (see example below.)

6.6.3 Example: Two interacting Neuron Pools

The code in this subsection simulates two pools of leaky-integrate-and-fire neurons, which interact mutually. It duplicates the variables and code from the previously used inf.c example program, but adds some code for the interaction and its control slider in the GUI.

The program is shown because it demonstrates how to write code using the macros explained in the previous section 6.6.2 that can either be compiled serially with GUI, but for parallel execution using MPI (or MPI/OpenMP) as well. The advantage would be that one can conveniently test a small version of the program with GUI, but run scaled-up large versions on a parallel computer without changing a single line of code. Both versions could even use the same environment files for parameter settings.

The idea is to cut the serial code into pieces that can be distributed across a number of MPI processes. The RANK() or COND()-macros are then used to select the respective code bits for execution in the individual processes. In order to set up the model properly one has to exchange data computed in one thread but needed in others, too. This is done by calls to the connect()-function in a top-level routine fmpi_connections(), see section 6.4.

The RANK, COND, and CONNECT-macros expand (basically) into empty code if the so prepared program is compiled serially. Using the macros appropriately, possibly in conjuction with the other macros in sections 6.4 and 6.3, can result in code that can be compiled serially and for parallel execution.

Here is one such magic codes (some parts have been cut out (mainly things related to display and output); the full source code should be in the Felix expl/parallel directory):

```
// infpairmpi.c

# include <felix.h>

# define N 100        /* number of neurons     */
# define tau 10.      /* membrane time const.  */
Vector  pot1, pot2;   /* potentials            */
Matrix  J1, J2;       /* connections           */
bVector o1, o2;       /* vector of spikes      */
Vector  v1, v2;       /* for help              */
int stp=0;

...

BEGIN_DISPLAY
....


BEGIN_OUTPUT
....


void fmpi_connections()
{
  CONNECT( 0, o1, bVECTOR, N, 1, o1 );
  CONNECT( 1, o2, bVECTOR, N, 0, o2 );
}


int main_init()
{
  randomize( time(NULL) + 100*myrank ); // not sure this safe ???????????????
  SET_STEPSIZE( .5 )

  RANK(0)
  {
    J1   = Get_Matrix( N, N );
    pot1 = Get_Vector( N );
    v1   = Get_Vector( N );
  }
  o1   = Get_bVector( N );

  RANK(1)
  {
    J2   = Get_Matrix( N, N );
    pot2 = Get_Vector( N );
    v2  = Get_Vector( N );
  }
  o2  = Get_bVector( N );
}

int init()
{
  int i;

  RANK(0)
  {
    Clear_bVector(N,o1);
    Clear_Vector(N,v1);
    for (i=0; i<N; i++)
      pot1[i] = equal_noise();   // random initialisation
    Make_Matrix( N, N, J1, 1./N , .4/N );
  }

  RANK(1)
  {
    Clear_bVector(N,o2);
    Clear_Vector(N,v2);
    for (i=0; i<N; i++)
      pot2[i] = 0;       // no random initialisation !
    Make_Matrix( N, N, J2, 1./N , .4/N );
  }

  stp=0;
}

int step()
{
  int i;

  RANK(0)
  {
    for (i=0;i<N;i++)
      leaky_integrate ( tau, pot1[i],
             0.01*( sinput + sJ*v1[i] + sJc*o2[i]
               + snoise*gauss_noise()
               )
                    );
    Fire_Reset( N,  pot1, 1.0, 0.0, o1 );
    bMult( N, N, J1, o1, v1 );
  }

  RANK(1)
  {
    for (i=0;i<N;i++)
      leaky_integrate ( tau, pot2[i],
             0.01*( sinput + sJ*v2[i] + sJc*o1[i]
               + snoise*gauss_noise()
               )
                    );
    Fire_Reset( N,  pot2, 1.0, 0.0, o2 );
    bMult( N, N, J2, o2, v2 );
  }

  stp++;

  MPI_ONLY(  // this ensures we don't run forever on the cluster
    if (stp >= 500)
    {
      MPI_Finalize();
      exit(0);
    }
  )
}
```

More explanations????????

The serial version of the code is compiled with “Felix infpairmpi” and run with "infpairmpi" from the command line as usual. The GUI should pop up as for standard serial Felix applications. If data storage is switched on, data of the first pool is written to file "pot1". Data of the second pool is not stored. The simulation runs until it is killed in the GUI.

The parallel version is compiled with "pFelix infpairmpi" and, e.g., run with "mpirun -np 2 infpairmpi" (It might be that you have to use other ways to run programs on your parallel computer, e.g., if the system adminstrator requires using a job scheduler). The parallel executable will not pop up a GUI. Data of the first pool will be written to "pot1-0" (by the first process); data of the second pool will not be saved, because no output files have been declared for the second process. The simulation exits after a certain number of steps (500).

Chapter 7

Example Programs

7.1 Leaky-Integrate-and-Fire Neural Network

```
/* Example-program: inf.c  */

# include <felix.h>

# define N   100    /* number of neurons      */
# define tau  10.   /* membrane time constant */

float I = 1.1,      /* Common input to units  */
      J0 = 1.1,     /* Coupling strength      */
      sigma = .1;   /* noise level            */

Vector  x;          /* potentials             */
Matrix  J;          /* connections            */
bVector z;          /* vector of spikes       */
Vector  v;          /* auxiliary variable     */


NO_DISPLAY

NO_OUTPUT


int main_init()
{
  /* init. random number generator and stepsize */
  randomize( time(NULL) );
  SET_STEPSIZE( .1 )

  /* allocate vectors and matrices */
  J = Get_Matrix( N, N );
  x = Get_Vector( N );
  z = Get_bVector( N );
  v = Get_Vector( N );
}

int init()
{
  int i;

  Clear_bVector(N,z);
  Clear_Vector(N,v);

  /* init. potentials with random values between 0 and 1 */
  for (i=0; i<N; i++)
    x[i] = equal_noise();

  /* init. J with gaussian distr. random numbers */
  Make_Matrix( N, N, J, 1.0/N, .4/N  );
}

int step()
{
  int i;

  for (i=0;i<N;i++)  // leaky integration for all neurons
    leaky_integrate ( tau, x[i],
                  I + J0*v[i] + sigma*gauss_noise() );

  Fire_Reset( N,  x, 1.0, 0.0, z );  // firing and reset

  bMult( N, N, J, z, v ); // redistribution of spikes
}
```

7.2 Coupled Chaotic Roessler Oscillators

Integrates differential equations with Runge-Kutta

Uses xy-plots

```
/*
 *  roessler.c -- coupled chaotic Roessler oscillators
 *                or  asymmetric damped harmonic oscillators
 */

#include "felix.h"


# define STEPSIZE .01
float t;

# define N 64       /* number of units     */
# define n  3       /* order of diff.system */
```

```
Vector  x;      /* x1 ... xN, y1 .... yN, z1 .... zN */
Vector  dxdt;     /* derivatives */
Vector domega;   /* used to give oscillators a gradient in properties */
Matrix  J;         /* connections  (if not meanfield couplings)  */
                   /* diffusive or random ....                   */

Vector  cfields;   /* coupling fields; either meanfield or diffusive
                      or random connectivity */
float xx1, yy1;

SwitchValue sosc = OFF;  /* Roessler or damped harmonic oscillators */
SwitchValue smean = ON;       /* mean field coupling */
SwitchValue sdiffusive = OFF; /* diffusive coupling */
SwitchValue swrand = OFF;    /* random connections */

SliderValue somega = 1000;
SliderValue sdelomega = 100;
SliderValue sepsilon = 100;
SliderValue sa = 150;


BEGIN_DISPLAY

 SWITCH( "osci type", sosc )
 SWITCH( "mean", smean )
 SWITCH( "diffusive", sdiffusive )
 SWITCH( "random", swrand )

 SLIDER( "mean omega", somega, 500, 1500)
 SLIDER( "delta omega", sdelomega, 0, 500)
 SLIDER( "coupling strength", sepsilon, 0, 500)
 SLIDER( "a", sa, 0, 500)

 WINDOW("signals")

  RASTER( "x", AR, AC, x, VECTOR,  N, 0, 0.0, 1.0, 2)

 WINDOW("MF-xy-plot")

  PLOT("x-y", AR, AC, &xx1, VECTOR, 1, 0, 0, 0, -20., 20.,
                      &yy1, VECTOR, 1, 0, 0, 0, -20., 20.,  2 );

 WINDOW("xy-plot")

  PLOT("x-y", AR, AC, x, VECTOR, N, n, 0, 0, -20., 20.,
                      x, VECTOR, N, n, 0, 1, -20., 20.,  2 );

 WINDOW("x(t)")

  GRAPH( "x1", AR, AC, x, VECTOR, N, 0, 0, 0, -20, 20 )
  GRAPH( "x2", AR, NC, x, VECTOR, N, 0, 1, 0, -20, 20 )
  GRAPH( "y1", NR, CO, &x[N], VECTOR, N, 0, 0, 0, -20, 20 )
  GRAPH( "y2", AR, NC, &x[N], VECTOR, N, 0, 1, 0, -20, 20 )

  GRAPH( "z1", NR, CO, &x[2*N], VECTOR, N, 0, 0, 0, 0., 20 )
  GRAPH( "z2", AR, NC, &x[2*N], VECTOR, N, 0, 1, 0, 0., 20 )

 WINDOW("MF")

  GRAPH( "x1", AR, AC, &xx1, VECTOR, 1, 0, 0, 0, -20., 20.)
  GRAPH( "x2", AR, NC, &yy1, VECTOR, 1, 0, 0, 0, -20., 20.)

END_DISPLAY

NO_OUTPUT


int main_init()
{

  SET_STEPSIZE( STEPSIZE )
  randomize( time(NULL) );

  J   = Get_Matrix(N,N);
  x   = Get_Vector(N*n);
  dxdt= Get_Vector(N*n);
  domega = Get_Vector(N);
  cfields = Get_Vector(N);
}


int init()
{
  int i;

  Clear_Vector(N, domega);
  Clear_Vector(N, cfields);
  Clear_Vector(N*n, x);
  Clear_Vector(N*n, dxdt);
  t = 0.0;

  for (i=0 ; i<N; i++)
  {
    domega[i] = -.5+(1.*i)/N;
    cfields[i] = 0.0;
    x[i] = 4.;
    x[N+i] = 4;
  }

  Clear_Matrix( N,N, J);
  Make_Matrix( N, N, J, 1, 1);
}
```

```
void derivs(x,y,dfdx)
float x;
float *y;
float *dfdx;
{
  int i,j;
  float omeg,a;

  a = .001*sa;

  for (i=0;i<N;i++)
  {
    if (sosc)  /* original roessler */
    {
      omeg = .001*(somega + sdelomega*domega[i]);
      dfdx[i]     = -omeg*y[N+i] - y[2*N+i] + cfields[i];
      dfdx[N+i]   = omeg*y[i] + a*y[N+i];
      dfdx[2*N+i] = .4+y[2*N+i]*(y[i]-8.5);
    }
    else   /* antisymm. undamped harm.osc. */
    {
      omeg = .001*(somega + sdelomega*domega[i]);
      dfdx[i]     =    a*y[i] -omeg*y[N+i] - y[2*N+i] + cfields[i];
      dfdx[N+i]   =  omeg*y[i] +   a*y[N+i];
      dfdx[2*N+i] = .4+y[2*N+i]*(y[i]-8.5);
    }
  }
}


int step()
{
  int i;
  float mf,epsfac;
  static float tlast=-1,phi1;


  rk4(x, dxdt, N*n, t, step_size, x, derivs);


  epsfac = .001*sepsilon;

  if(swrand)  /* different amplitude scaling in alternatives ... */
  {
    Mult( N, N, J, x, cfields );   /* good luck; first components of
                                      systems are first N vals of x */
    epsfac /= N;
  }
  else if(sdiffusive) /* open boundaries */
  {
    cfields[0] = x[1]-x[0];
    for(i=1;i<N-1;i++)
     cfields[i] = x[i+1]+x[i-1]-2*x[i];
    cfields[N-1] = x[N-2]-x[N-1];
  }
  else if (smean) /* mean field */
  {
    mf = Sum(N, x)/(float)N;
    for(i=0;i<N;i++)
     cfields[i] = mf;
  }
  else
  {
    for(i=0;i<N;i++)
     cfields[i] = 0.;
  }

  for(i=0;i<N;i++)   /* scale with coupling strength */
    cfields[i] *= epsfac;

  xx1 = Sum( N, x)/N;
  yy1 = Sum( N, &x[N])/N;

  t+=step_size;
}
```

7.3 Homogeneous Fields

```
/* ei-field.c -- two-dimensional excitatory/inhibitory neural field model
 *                probabilistic spiking neurons
 *                stimulus is a single long moving bar or two bars moving
 *                in parallel or antiparallel
 */


# include <felix.h>

# define tau1  3.
# define tau2  5.0

long stp = 0;
float sim_time, noise_fac;

Layer       input,
            pot1, pot2,
            f1, f2;

SpikeLayer  out1, out2;

# define L_SIZE11   8.0       /* FWHM in columns (float) */
# define M_SIZE11   8         /* Kernel dimension (int)  */
# define FM_SIZE11  (2*M_SIZE11+1)

# define L_SIZE12   8.0       /* FWHM in columns (float) */
# define M_SIZE12    4         /* Kernel dimension (int)  */
# define FM_SIZE12  (2*M_SIZE12+1)

# define L_SIZE21   8.0       /* FWHM in columns (float) */
# define M_SIZE21    4        /* Kernel dimension (int)  */
# define FM_SIZE21  (2*M_SIZE21+1)


UniKernel   kernel11,
            kernel12,
            kernel21;

Layer       link11,
            link12,
            link21;

# define barlength  25
# define barskip     0    /* 5  */
# define barsigma    7
# define BARINITOFFS 14.

double  yy1, yy2;
int bardirection = 1;


SwitchValue santi = OFF;
SwitchValue scent = OFF;

SliderValue sI1     = 85;
SliderValue sI2     = 85;
SliderValue sI     = 85;
SliderValue snoise = 20;
SliderValue sJ11  = 100;
SliderValue sJ12  = 40;
SliderValue sJ21  = 600;
SliderValue sspeed = 0;

BEGIN_DISPLAY

 SWITCH( "anti", santi)
 SWITCH( "center", scent )

 SLIDER( "Signal Input", sI, 0, 1000 )
 SLIDER( "E ", sI1, -200, 200 )
 SLIDER( "I ", sI2, -200, 200 )
 SLIDER( "noise", snoise, 0, 1000 )
 SLIDER( "J11", sJ11, 0, 500)
 SLIDER( "J12", sJ12, 0, 300)
 SLIDER( "J21", sJ21, 0, 1000)
 SLIDER( "speed", sspeed, 0, 1000);

 WINDOW("Excitation")

  IMAGE( " input ", AR, AC, input, LAYER, xsize, ysize, 0.0, 2.1, 1)
  IMAGE( "  pot1  ", AR, NC, pot1, LAYER, xsize, ysize, -.5, 1.0, 1)
  IMAGE( "  out1  ", NR, CO, out1, SPIKE_LAYER, xsize, ysize, 0.0, 1.0, 1)

WINDOW("Inhibition")

  IMAGE( " input ", AR, AC, input, LAYER, xsize, ysize, 0.0, 2.1, 1)
  IMAGE( "  pot2  ", AR, NC, pot2, LAYER, xsize, ysize, -.5, 1.0, 1)
  IMAGE( "  out2  ", NR, CO, out2, SPIKE_LAYER, xsize, ysize, 0.0, 1.0, 1)

WINDOW("Kernels")

  IMAGE( " k11", AR, AC, kernel11, CONSTANT LAYER,
               FM_SIZE11, FM_SIZE11, 0.0, 1., 5)
  IMAGE( " k12", NR, AC, kernel12, CONSTANT LAYER,
               FM_SIZE12, FM_SIZE12, 0.0, 1., 5)
  IMAGE( " k21", NR, AC, kernel21, CONSTANT LAYER,
               FM_SIZE21, FM_SIZE21, 0.0, 1., 5)

END_DISPLAY


BEGIN_OUTPUT

 OUTFILE("phi1")

  SET_SAVE_FILE_FLAG( THISFILE, ASCII, ON)
  SAVE_VARIABLE( "phi1 (pot1)", pot1, MATRIX, xsize, ysize,  SKIP | GRID ,
                 TSkip(2), Grid(26, xsize, 100, 32, ysize, 100) )

 OUTFILE("phi2")

  SET_SAVE_FILE_FLAG( THISFILE, ASCII, ON)
  SAVE_VARIABLE( "phi2 (pot2)", pot2, MATRIX, xsize, ysize,  SKIP | GRID ,
                 TSkip(2), Grid(38, xsize, 100, 32, ysize, 100) )

END_OUTPUT


static void init_bars(centerflag)
int centerflag;
{
  if (centerflag)  /* center */
  {
    yy1 = yy2 = ysize/2;
  }
  else
  {
    yy1 = BARINITOFFS;
    if (santi)
      yy2 = ysize-BARINITOFFS;
    else
      yy2 = BARINITOFFS;
    bardirection = 1;
  }
}

static void move_bars()
{
  if (scent)
  {
    init_bars(1);
    return;
  }

  if (yy1 > ysize-BARINITOFFS ||
      yy1 < BARINITOFFS)
    bardirection *= -1;

  yy1 += .001*bardirection*sspeed;
  if (santi)
    yy2 -= .001*bardirection*sspeed;
  else
    yy2 += .001*bardirection*sspeed;
}



static void smooth_bars( out )
Matrix out;
{
  int i, j, s1, s2, s3, s4;
  static double fac=0;
  double h;

  if (fac==0) fac = -.5/(float)(barsigma*barsigma);

  s2 = (xsize - barskip)/2;
  s1 = s2-barlength;
  s3 = (xsize + barskip)/2;
  s4 = s3 + barlength;

  for (j = 0; j<ysize; j++)
  {
    h = elem( out, j, s1, xsize) = triangle( fac * (yy1-j)*(yy1-j));
    for (i=s1+1; i<s2; i++)
      elem( out, j, i, xsize) = h;

    h = elem( out, j, s3, xsize) = triangle( fac * (yy2-j)*(yy2-j));
    for (i=s3+1; i<s4; i++)
```

```
      elem( out, j, i, xsize) = h;
  }
}



int main_init()
{
  int i;

  randomize(  time(NULL) );

  input = Get_Layer();
  pot1  = Get_Layer();
  f1    = Get_Layer();
  out1  = Get_SpikeLayer();
  pot2  = Get_Layer();
  f2    = Get_Layer();
  out2  = Get_SpikeLayer();

  link11 = Get_Layer();
  link12 = Get_Layer();
  link21 = Get_Layer();

  kernel11 = Get_UniKernel( FM_SIZE11, FM_SIZE11 );
  kernel12 = Get_UniKernel( FM_SIZE12, FM_SIZE12 );
  kernel21 = Get_UniKernel( FM_SIZE21, FM_SIZE21 );

  Set_Circ_Func_Uni_Kernel( kernel11, FM_SIZE11, FM_SIZE11, gaussian,
                            1., L_SIZE11, 0. );
  Set_Circ_Func_Uni_Kernel( kernel12, FM_SIZE12, FM_SIZE12, gaussian,
                            1., L_SIZE12, 0. );
  Set_Circ_Func_Uni_Kernel( kernel21, FM_SIZE21, FM_SIZE21, gaussian,
                            1., L_SIZE21, 0. );

  SET_STEPSIZE(0.5);
  noise_fac = sqrt(24.0/step_size);
}



int init( )
{
  int i,j;

  stp = 0;

  Clear_Layer(input);
  init_bars( scent );
  smooth_bars( input );

  Clear_Layer(pot1);
  Clear_SpikeLayer(out1);

  Clear_Layer(pot2);
  Clear_SpikeLayer(out2);
}

int step()
{
  int i,j,k;

  if (stp >= 36050)
    exit (0);

  /********************/
  /* compute stimulus */
  /********************/

  move_bars();
  smooth_bars( input );

  /********************/
  /* compute dynamics */
  /********************/

  /* excit. units */

  for (i=0; i<ysize; i++)
  {
    for (j=0; j<xsize; j++)
    {
      leaky_integrate( tau1, elem( pot1, i, j, xsize) ,
           0.001*(sI1 + sI*gauss_noise()*elem( input , i, j, xsize)
              + sJ11*elem( link11,i,j, xsize)
              - sJ12*elem( link12,i,j, xsize)
              + (snoise*noise_fac)*(equal_noise() - 0.5) ) ) ;
      elem( f1, i, j, xsize) = RAMP( elem( pot1, i, j, xsize) );
      elem( out1, i, j, xsize) = PROB_FIRE( elem( f1, i, j, xsize) );

    } /* END j */

    for (j=0; j<xsize; j++)
    {
      leaky_integrate( tau2, elem( pot2, i, j, xsize) ,
           0.001*( sI2 +  sJ21*elem( link21,i,j, xsize)
                 + (snoise*noise_fac)*(equal_noise() - 0.5) ) ) ;
      elem( f2, i, j, xsize) = RAMP( elem( pot2, i, j, xsize) );
      elem( out2, i, j, xsize) = PROB_FIRE( elem( f2, i, j, xsize) );
    } /* END j */
  } /* END i */


  bConvolute_2d_Uni( out1, kernel11, xsize, ysize, FM_SIZE11, FM_SIZE11, link11);
  bConvolute_2d_Uni( out1, kernel21, xsize, ysize, FM_SIZE21, FM_SIZE21, link21);
  bConvolute_2d_Uni( out2, kernel12, xsize, ysize, FM_SIZE12, FM_SIZE12, link12);

  stp++;

} /* END of step() */
```

Appendix A

Installation Guide

This appendix describes how to install the Felix simulation tool on serial and parallel computers. Lacking free time I never implemented proper autoconfiguration facilities. Therefore installation is quite low-level. However, a number of people have been able to install Felix on serial Linux boxes following the instructions below. Windows/Cygwin installations as well as installation of the parallel Felix extension can be a little more tricky.

The first part of this appendix describes the installation of the serial Felix version. This by default comprises the graphical user interface. Compiling Felix for parallelised code is described in the 2cd section. If you plan to use MPI, the GUI will not be available. The graphics works, however, with the SSE-BLAS and OpenMP code.

The following assumes that $FELIXDIR is the top-level directory of your Felix installation.

There should be a number of subdirectories (after unpacking)

$FELIXDIR/src : Source code of Felix kernel routines and libraries

$FELIXDIR/xview : Sorce code of X11 extensions used for the Felix-GUI

$FELIXDIR/lib : Felix libraries (created during compilation)

$FELIXDIR/expl : A number of example applications

$FELIXDIR/tools : A number of tools to transform Felix data files (e.g., for creating raster plots and gifs)

To compile the Felix core only the code in $FELIXDIR/src is needed. If you want the GUI you need in addition the code in $FELIXDIR/xview. These directories comprise several relevant Makefiles

$FELIXDIR/src/Makefile : main source code (compilation of serial lib libf)

$FELIXDIR/xview/Makefile : graphics extensions for X11 (compilation of serial lib libxf)

$FELIXDIR/Makefile : master Makefile to compile a serial application (envoked by the "Felix" command)

$FELIXDIR/src/Makefile.parallel : main source code (compilation of parallel lib libpf)

$FELIXDIR/Makefile.parallel : master Makefile to compile a parallel application (envoked by the "pFelix" command)

The first three Makefiles are required for compiling the serial libraries and code; the second two for parallel libs and code.

A.1 Standard (serial) Installation

A.1.1 Prerequisites

The Graphical user interface is built on X11 and a pretty old Widget tool called XView. XView is used for historical reasons. It was originally developed by Sun Microsystems who ceased supporting it in about 1995, when Motif became more dominant. It is still possible to get XView sources and binaries, but this gets more and more difficiult (in particular I don't know of any 64 bit packages).

Compilation of Felix presupposes an installed X11R6 package assumed to be in the standard location: /usr/X11R6 . X11R6 is by default contained in virtually all Linux installations. If this is the wrong path it has to be corrected in the Makefiles, i.e., those in ../src, ../xview and the top-level makefile.

The Felix GUI further requires installed XView libraries *libolgx* and *libxview*, e.g., in /user/openwin/lib

Felix further requires the XView development kit for include files, e.g., in /usr/openwin/include

It is possible to set an environment variable OPENWINHOME pointing at the location of the XView libs and include files during compilation.

Redhat/SuSe/Cygwin users: An XView rpm can be downloaded here http://www.physionet.org/physiotools/xview/

Ubuntu/Kubuntu/Debian users: The XView packages are in some (K)ubuntu repositories.

A.1.2 Serial Felix Installation

1. Create the Felix top-level directory ($FELIXDIR) where you want it.

   Default would be something like $HOME/felix.

2. Goto the target directory $FELIXDIR and unpack and untar sim.tar.gz **in** it by calling "tar -xzf sim.tar.gz"

3. Set environment variables for your shell. For the bash-shell (default in many Linuxes), add the following in $HOME/.bashrc :

```
export OPENWINHOME="/usr/openwin"
export FELIXDIR="\$HOME/felix"
export LD_LIBRARY_PATH="\$FELIXDIR/lib:/usr/X11R6/lib:\$LD_LIBRARY_PATH"
alias Felix="\$FELIXDIR/Felix"
```

The precise locations of the directories in the above exports possibly need to be adapted to your own file hierarchy. It might also be that /usr/X11R6/lib is already in your path or that the libs it contains are accessible by other means (in that case you can omitt it in the export above).

Beside that make sure "." (current directory) is in PATH (type echo $PATH in a shell and look for it). If it is not there you will have to type ./<progname> to run programs. Just <progname> would fail with “permission denied” or “program not found” or a similar error message.

4. Dont forget to execute “source .bashrc” in your running (bash-)shell after setting the environment variables. Alternatively, you can start a new shell so that the environment variables get set correctly.

5. Run “make install” in $FELIXDIR .

   If everything goes well this should compile the source code in $FELIXDIR/src and $FELIXDIR/xview, create the respective Felix core and GUI libraries, and move them to $FELIXDIR/lib.

   If this step is successfull you will have the (serial) Felix libraries *libxf* and *libf* in $FELIXDIR. Otherwise something went wrong.

6. Test a Felix example in $FELIXDIR/expl, e.g., inf.c :

   (a) change to the directory $FELIXDIR/expl
   (b) run “Felix inf” : the program “inf” should be compiled
   (c) run “inf” : “inf” should run and the graphical interface pop up

   If the test runs successful, you are ready to use the serial Felix version. Check out the examples in $FELIXDIR/expl .

A.1.3 Additional Notes

1. If you try to compile a felix program and get an error message that *panel.h, frame.h* or so are not found, then you don't have XView installed properly or haven't set the proper paths in the Makefiles.

A.2 Installation of Parallel Felix

The parallel Felix extensions are experimental code. Whereas much of the serial code (but not all) has been used for research for already many years, the parallel code is much more recent. I can't give much advise on it, it is in a pretty chaotic state, and it probably contains bugs .... feel free to improve it. Send patches or error warnings ...

Felix implements 3 levels of parallelism, which can at least intentionally be used simultaneously in any mix (this is mostly untested):

BLAS : Given proper BLAS/ATLAS libraries you might be able to use the SSE extensions of Intel and AMD CPUs. Note that you can use BLAS routines even if you do not have a multi processor system. BLAS routines support highly optimised Matrix/Vector Math. Some BLAS versions support automatic threading if you are on a multiprocessor SMP machine (e.g. gotoBLAS and, I believe, Intel MKL BLAS too). This, however, might interfere with level 2 OpenMP parallelism. If you are not careful, each OpenMP thread might spawn a number of BLAS threads. The BLAS libs usually support environment variables or other means to control the number of spawned threads.

OpenMP : OpenMP is a simple framework to parallelise outer loops on SMP multiprocessor machines. It automatically spawns threads that distribute separate parts of the loop over the available processors. Although simple to use OpenMP is suboptimal in various respects as compared to hand-coded threaded code. However, I have seen nice speed-ups for some of applications. gcc will support OpenMP from version 4.2 upward; the Intel compiler also implements the OpenMP standard. Since gcc isn't officially out yet, I use hacks to compile the OpenMP-parallel Felix code with icc, the Intel compiler. That makes some of the Makefiles look pretty nasty... (I have also compiled a pre-released gcc-4.2 snapshot. Seems to work, too.)

MPI : MPI is a message passing standard for multi processor systems including Symmetric Mulit Processors (SMPs) and Beowulf computer clusters. Felix uses very few very simple constructs to transport data between several co-operating processes in distributed Felix programs (see fmpi.c/h). In principle these are vectors/matrices transported between variables local to each process. Each process is running the same program but has a certain "rank" which can be used in the code to make parts of it selectively executable on some processes only. Check the paralle examples in $FELIXDIR/expl for more details.

The parallel version has its own Makefiles $FELIXDIR/src/Makefile.parallel and $FELIXDIR/Makefile.parallel which compile Felix versions without graphical interfaces. They contain flags for activating the different options.

You might also want to use these flags in the serial Makefiles. In that case you need to adapt the compiler settings and if you choose to activate MPI, you have to switch the graphical user interface off. BLAS and OpenMP parallelism, however, is comaptible with the GUI.

A.2.1 Prerequisites

You do not need a parallel computer to experiment with the parallel extensions. Each modern Intel or AMD CPU supports the SSE2 vectorisation which you may use in your BLAS version. You can also install and use MPI and OpenMP compilers/code on a serial machine. This way you can write and test code on, e.g., your laptop, before going on a bigger machine.

BLAS : A proper BLAS implementation, ie. ATLAS or gotoBLAS. The default BLAS that comes with many Linux versions is probably not speed-optimised (meaning that you can loose tremendous speed benefits for some matrix/matrix and matrix/vector operations. [At the moment BLAS is only used for some Felix functions — don't expect too much.]

OpenMP : As long as gcc 4.2 isn't available, you need another OpenMP capable compiler. There are some open source versions (I have used OmniMP, but wasn't happy with its optimisation capabilities). I now use the Intel compiler, which has a free licence for single academic users. Thanks to Intel for that! You can also compile a prerelease of gcc 4.2 (or higher). This has the OpenMP standard built in. You need to adapt the Makefiles in that case.

MPI : The Felix MPI version works only without the graphical interface. It was developed for a computer cluster on which graphical interfaces make little sense. You can potentially compile with GUI in which case I would suspect each MPI process tries to open its own GUI. I never tested this.

You need gcc or icc or another C compiler and an MPI library. I use mostly MPICH(1) but at least previous parallel Felix versions worked also with LAM. I haven't checked MPICH(2) so far, but there is little reason why it should not work (one hears communication is considerably faster than MPICH(1)).

Note that Intel provides its own MPI libs, but I don't have them. Might be a useful investigation: Although I use icc, I link against the mpich libraries. That requires rather uncomfortable compiler settings (see Makefile.parallel).

One can run into problems with the MPI runtime environment not finding dynamic libraries. I therefore link part of the libs statically. That makes programs bigger. Alterantively, there are also linker switches to tell executables where to find the libs.

I use icc because to combine MPI with OpenMP one (obviously) needs an OpenMP capable compiler. Using Intel to date is the only (more or less) tested case (I have also testet a prerelease of gcc-4.2 very briefly; seems to work in principle). The Makefile.parallel is for icc, so have a look into it. You will see that I don't use the usual MPI compiler wrapper script, mpicc, but supply include and library directories etc directly to icc. You can probably avoid this, if you compile your own MPICH (or LAM?) using icc and use the mpicc version generated this way. I DO, however, use the "mpirun"-script of the MPICH standard installation.

A.2.2 Compilation of Parallel Felix

Compilation of parallel Felix follows the same steps as for the serial version. The instructions below compile a parallel library *libpf*, which can coexist with the serial libraries as compiled in the first section of this appendix (*libf* and *libxf*). You only have to use the script pFelix to compile a parallel application code against the right parallel libs.

To compile a parallel version of Felix without graphical user interface follow these instructions:

1. Beside the environment variables for the serial version you need to add another one for the parallel Felix script. In your .bashrc add

   alias pFelix="$FELIXDIR/pFelix"

2. Enable the desired flags in the parallel Makefile in the src and/or main directories:

   BLAS : Just enable -DWITH_BLAS in src/Makefile.parallel. [BLAS should work with and without graphical user interface, so that you could also use the serial Makefile if you want the GUI (doesn't work, though, if -DWITH_MPI is also set)].

   I did occassionally have some problems with linking against the right libs. You might have to adapt the Makefiles to get BLAS working

OpenMP : To use OpenMP switch -DWITH_OMP on in the Makefile and adapt it to use your OpenMP capable compiler (and linker, and archiver). The graphical interface should work with OpenMP, so that you can use the serial Makefiles, if you want graphical output.

MPI : To use MPI activate -DWITH_MPI and -DNO_GRAPHICS in the Makefile.

3. Adapt compiler, linker, archiver and flags, paths and libs in the Makefiles as necessary.
4. Delete any old object files present from compiling serial libs earlier by evoking “make clean” from a shell
5. Compile the parallel libs with “make -f Makefile.parallel par” in the src-directory.

   This should produce a library libpf.a in the lib-directory
6. You link against the parallel library libpf.a automatically if you use the “pFelix” script for compilation of your application code. This requires proper settings in the top-level Makefile.parallel.
7. Test an example from $FELIXDIR/expl/parallel, i.e., compile it using “pFelix prog” and run the generated executable using, e.g., *mpirun -np 2 prog*, where “prog” is the base program name (i.e., infmpi).

Note that serial programs and parallel code that uses MPI are not (in general) compatible. You need, e.g., to declare in the parallel code, which buffers are transported between processes. I will describe elsewhere how you can write applications that can be compiled parallel and serial (with GUI), and use even the same environment files.

A.2.3 Additional Notes

1. There is a compiler flag -DTIMING in the source Makefile. If this is switched on during compilation, timing information for the main parts of a Felix programm will be printed for each individual process.
2. If necessary, you can link Intel libs statically using -i-static flag of icc; this acts more specific than -static which links everything statically
3. You can tell a binary where to expect a library, e.g., mpiCC -Wl,-rpath=$INTEL/cc/9.0.030/lib/ -o mpitest mpitest.C

A.3 Windows / Cygwin

The serial Felix versions runs properly under Cygwin and the Windows operating system. It is considerably slower than under Linux, but still usable for, e.g., presentations.

An XView rpm can be downloaded here (together with instructions of how to install Cygwin and XView): http://www.physionet.org/physiotools/xview/

There is no obvious reason why the parallel Felix extensions should not work given the right tools (MPI, OpenMPI, BLAS). However, it has never be tried to compile parallel Felix on a Windows box.